MSDN C# 编程指南 & 参考手册 2015

wizardforcel

Published with GitBook

目錄

MSDN C# 编程指南 & 参考手册 2015

C# 编程指南

本节提供有关关键的 C# 语言功能和 C# 可通过 .NET Framework 访问的功能的详细信息。

本节中大部分内容都假定您已了解有关 C# 和一般编程概念的一些知识。如果您刚开始学习编程或 C#，您可能还希望访问 C# Developer Center（C# 开发人员中心），此处提供很多教程、示例和视频，可帮助您入门。

有关特定的关键字、运算符和预处理器指令的信息，请参见 C# 参考。有关 C# 语言规范的信息，请参见 C# 语言规范。

语言部分

不安全代码和指针（C# 编程指南）

XML 文档注释（C# 编程指南）

平台部分

Application Domains (C# and Visual Basic)

程序集和全局程序集缓存（C# 和 Visual Basic）

特性（C# 和 Visual Basic）

集合（C# 和 Visual Basic）

异常和异常处理（C# 编程指南）

文件系统和注册表（C# 编程指南）

互操作性（C# 编程指南）

反射（C# 和 Visual Basic）

重要章节

http://go.microsoft.com/fwlink/?LinkId=195406

http://go.microsoft.com/fwlink/?LinkId=195407

请参阅

C# 参考

Visual C#

在 C# 程序内部

本节讨论 C# 程序的一般结构，还包括标准的“Hello, World!”示例。

本节内容

- Hello World -- 您的第一个程序（C# 编程指南）
- C# 程序的通用结构（C# 编程指南）

相关章节

- Visual C# 入门
- C# 编程指南
- C# 参考
- <paveover>C# Sample Applications

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

Hello World -- 您的第一个程序（C# 编程指南）

以下过程创建 C# 版本的传统“Hello World!”程序。该程序显示字符串 Hello World!

有关入门概念的更多示例，请参见 Visual C# 和 Visual Basic 入门。

注意
以下说明中的某些 Visual Studio 用户界面元素在计算机上出现的名称或位置可能会不同。 这些元素取决于你所使用的 Visual Studio 版本和你所使用的设置。有关详细信息，请参阅个性化 Visual Studio IDE。

创建并运行控制台应用程序

1. 启动 Visual Studio。
2. 在菜单栏上，选择“文件”，“新建、“项目”。

   将打开“新建项目”对话框。
3. 展开“已安装”，展开“模板”，展开“Visual C#”，然后选择“控制台应用程序”。
4. 在“名称”框中，指定项目名称，然后选中“确定”按钮。

   新项目出现在“解决方案资源管理器”中。
5. 如果 Program.cs 不是在“代码编辑器”中打开，则打开“解决方案资源管理器”中“Program.cs”的快捷方式菜单，然后选择“视图代码”。
6. 用下面的代码替换 Program.cs 的内容。

```
// A Hello World! program in C#.
using System;
namespace HelloWorld
{
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}
```

7. 选择 F5 键运行项目。命令提示窗口将显示，其中包含行 Hello World!

接着，检查本程序的重要部分。

注释

第一行包含注释。 // 字符将这行的其余内容转换为注释内容。

```
// A Hello World! program in C#.
```

还可通过将文本块置于 / 和 / 字符之间将其注释掉。这将在下面的示例中显示。

```
/* A "Hello World!" program in C#.
This program displays the string "Hello World!" on the screen. */
```

Main 方法

C# 控制台应用程序必须包含一个 Main 方法，用于控制程序的开始和结束。在 Main 方法中创建对象和执行其他方法。

Main 方法是驻留在类或结构内的 static（C# 参考）方法。在前面的“Hello World!”示例中，此方法驻留在一个名为 Hello 的类中。可以用下列方式之一声明 Main 方法：

- 该方式返回 **void**。

```
static void Main()
{
    //...
}
```

- 它还可以返回整数。

```
static int Main()
{
    //...
    return 0;
}
```

- 无论使用哪种返回类型，它都可以带有参数。

```
static void Main(string[] args)
{
    //...
}
```

- 或 -

```
static int Main(string[] args)
{
    //...
    return 0;
}
```

Main 方法中定义的参数args是一个string（字符串）类型的数组，该数组的内容为在命令行下调用本程序时提供的参数。 与 C++ 不同，数组不包含可执行 (exe) 文件的文件名。

有关如何使用命令行参数的更多信息，请参见 Main() 和命令行参数（C# 编程指南）中的示例和如何：使用命令行创建和使用程序集（C# 和 Visual Basic）。

通过按 F5 在调试模式下运行程序时，在 Main 方法的末尾调用 ReadKey 将使得控制台窗口无法关闭，从而使您可以阅读输出。

输入和输出

C# 程序通常使用 .NET Framework 的运行库提供的输入/输出服务。System.Console.WriteLine("Hello World!"); 语句使用 WriteLine 方法。此方法是运行库中的 Console 类的输出方法之一。它显示了标准输出流使用的字符串参数，输出流后面跟一个新行。其他 Console 方法用于不同的输入和输出操作。如果程序开始处包含 using System; 指令，则无需完全限定 System 类和方法即可直接使用它们。例如，您可以改为调用 Console.WriteLine 而非 System.Console.WriteLine：

```
using System;
```

```
Console.WriteLine("Hello World!");
```

有关输入/输出方法的更多信息，请参见 System.IO。

命令行编译和执行

可以使用命令行而不是 Visual Studio 集成开发环境 (IDE) 编译“Hello World!”程序。

从命令提示行编译并运行

1. 将前面过程的代码粘贴到任何文本编辑器中，并将文件保存为文本文件。文件 Hello.cs 的名称。C# 源代码文件使用的扩展名是 .cs。
2. 执行以下步骤之一打开命令提示符窗口：
   - 在 Windows 8 中，在“开始”屏幕，搜索开发人员命令提示，然后点击或选择“VS2012 开发人员命令提示”。

     将出现“开发人员命令提示符”窗口。
   - 在 Windows 7 中，打开“开始”菜单，展开当前 Visual Studio 版本的文件夹，打开“Visual Studio 工具”的快捷菜单，然后选择“VS2012 开发人员命令提示”。

     将出现“开发人员命令提示符”窗口。
   - 从标准“命令提示”窗口启用命令行生成。

     请参见 How to: Set Environment Variables for the Visual Studio Command Line。
3. 在命令提示窗口中，导航至包含 Hello.cs 文件的文件夹。
4. 输入下面的命令，编译 Hello.cs。

   csc Hello.cs

   如果您的程序中有没有编译错误，则将创建名为 Hello.exe 的可执行文件。
5. 命令提示符窗口中，输入以下命令运行程序：

   Hello

有关 C# 编译器及其选项的详细信息，请参阅 C# Compiler Options。

重要章节

在 Visual C# 2010 使用入门中编写 C# 程序

请参阅

C# 编程指南

在 C# 程序内部

字符串（C# 编程指南）

<paveover>C# Sample Applications

C# 参考

Main() 和命令行参数（C# 编程指南）

Visual C# 和 Visual Basic 入门

C# 程序的通用结构（C# 编程指南）

C# 程序可由一个或多个文件组成。每个文件都可以包含零个或零个以上的命名空间。一个命名空间除了可包含其他命名空间外，还可包含类、结构、接口、枚举、委托等类型。以下是 C# 程序的主干，它包含所有这些元素。

```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }

    class YourMainClass
    {
        static void Main(string[] args)
        {
            //Your program starts here...
        }
    }
}
```

相关章节

有关更多信息：

- 类（C# 编程指南）
- 结构（C# 编程指南）
- 命名空间（C# 编程指南）
- 接口（C# 编程指南）
- 委托（C# 编程指南）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

在 C# 程序内部

C# 参考

<paveover>C# Sample Applications

C# 编码约定（C# 编程指南）

C# 语言规范 未定义编码标准。但是，Microsoft 根据本主题中的准则来开发样本和文档。

编码约定可实现以下目的：

- 它们为代码创建一致的外观，以确保读取器专注于内容而非布局。
- 它们使得读取器可以通过基于之前的经验进行的假设更快地理解代码。
- 它们便于复制、更改和维护代码。
- 它们展示 C# 最佳做法。

命名约定

- 在不包括 using 指令的短示例中，使用命名空间限定。如果你知道命名空间默认导入项目中，则不必完全限定来自该命名空间的名称。如果对于单行来说过长，则可以在点 (.) 后中断限定名称，如下面的示例所示。

```
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- 你不必更改通过使用 Visual Studio 设计器工具创建的对象的名称以使它们适合其他准则。

布局约定

好的布局利用格式设置来强调代码的结构并使代码更便于阅读。Microsoft 示例和样本符合以下约定：

- 使用默认的代码编辑器设置（智能缩进、4 字符缩进、制表符保存为空格）。有关详细信息，请参阅选项、文本编辑器、C#、格式设置。
- 每行只写一条语句。
- 每行只写一个声明。
- 如果连续行未自动缩进，请将它们缩进一个制表符位（四个空格）。
- 在方法定义与属性定义之间添加至少一个空白行。
- 使用括号突出表达式中的子句，如下面的代码所示。

```
if ((val1 &gt; val2) && (val1 &gt; val3))
{
    // Take appropriate action.
}
```

注释约定

- 将注释放在单独的行上，而非代码行的末尾。
- 以大写字母开始注释文本。
- 以句点结束注释文本。
- 在注释分隔符 (//) 与注释文本之间插入一个空格，如下面的示例所示。

```
// The following declaration creates a query. It does not run
// the query.
```

- 不要在注释周围创建格式化的星号块。

语言准则

以下各节介绍 C# 遵循以准备代码示例和样本的做法。

String 数据类型

- 使用 **+** 运算符来连接短字符串，如下面的代码所示。

```
string displayName = nameList[n].LastName + ", " + nameList[n].
```

- 若要在循环中追加字符串，尤其是在使用大量文本时，请使用 StringBuilder 对象。

```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalal
var manyPhrases = new StringBuilder();
for (var i = 0; i &lt; 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

隐式类型的局部变量

- 当变量类型明显来自赋值的右侧时，或者当精度类型不重要时，请对本地变量进行隐式类型化。

```
// When the type of a variable is clear from the context, use v
// in the declaration.
var var1 = "This is clearly a string.";
var var2 = 27;
var var3 = Convert.ToInt32(Console.ReadLine());
```

- 当类型并非明显来自赋值的右侧时，请勿使用 var。

```
// When the type of a variable is not clear from the context, u
// explicit type.
int var4 = ExampleClass.ResultSoFar();
```

- 请勿依靠变量名称来指定变量的类型。它可能不正确。

```
// Naming the following variable inputInt is misleading.
// It is a string.
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- 避免使用 **var** 来代替 dynamic。
- 使用隐式类型化来确定 for 和 foreach 循环中循环变量的类型。

  下面的示例在 **for** 语句中使用隐式类型化。

```
var syllable = "ha";
var laugh = "";
for (var i = 0; i &lt; 10; i++)
{
    laugh += syllable;
    Console.WriteLine(laugh);
}
```

  下面的示例在 **foreach** 语句中使用隐式类型化。

```
foreach (var ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

无符号数据类型

- 通常，使用 **int** 而非无符号类型。 **int** 的使用在整个 C# 中都很常见，并且当你使用 **int** 时，更易于与其他库交互。

数组

- 当在声明行上初始化数组时，请使用简洁的语法。

```
// Preferred syntax. Note that you cannot use var here instead
string[] vowels1 = { "a", "e", "i", "o", "u" };

// If you use explicit instantiation, you can use var.
var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// If you specify an array size, you must initialize the elemer
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

委托

- 使用简洁的语法来创建委托类型的实例。

```
// First, in class Program, define the delegate type and a me
// has a matching signature.

// Define the type.
public delegate void Del(string message);

// Define a method that has a matching signature.
public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

```
// In the Main method, create an instance of Del.

// Preferred: Create an instance of Del by using condensed synt
Del exampleDel2 = DelMethod;

// The following declaration uses the full syntax.
Del exampleDel1 = new Del(DelMethod);
```

异常处理中的 try-catch 和 using 语句

- 对大多数异常处理使用 try-catch 语句。

```
static string GetValueFromArray(string[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        Console.WriteLine("Index is out of range: {0}", index);
        throw;
    }
}
```

- 通过使用 C# using 语句简化你的代码。如果你具有 try-finally 语句（该语句中 **finally** 块的唯一代码是对 Dispose 方法的调用），请使用 **using** 语句代替。

```
// This try-finally statement only calls Dispose in the finally
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}

// You can do the same thing with a using statement.
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset = font2.GdiCharSet;
}
```

&& 和 || 运算符

- 若要通过跳过必要的比较来避免异常和提高性能，请在执行比较时使用 && 来代替 &，使用 || 来代替 |，如下面的示例所示。

```
Console.Write("Enter a dividend: ");
var dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
var divisor = Convert.ToInt32(Console.ReadLine());

// If the divisor is 0, the second clause in the following conc
// causes a run-time error. The && operator short circuits wher
// first expression is false. That is, it does not evaluate the
// second expression. The & operator evaluates both, and causes
// a run-time error when divisor is 0.
if ((divisor != 0) && (dividend / divisor &gt; 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

New 运算符

- 隐式类型化时，请使用对象实例化的简洁形式，如下面的声明所示。

```
var instance1 = new ExampleClass();
```

上一行等同于下面的声明。

```
ExampleClass instance2 = new ExampleClass();
```

- 使用对象初始值设定项来简化对象创建。

```
// Object initializer.
var instance3 = new ExampleClass { Name = "Desktop", ID = 37414
    Location = "Redmond", Age = 2.3 };

// Default constructor and assignment statements.
var instance4 = new ExampleClass();
instance4.Name = "Desktop";
instance4.ID = 37414;
instance4.Location = "Redmond";
instance4.Age = 2.3;
```

事件处理

- 如果你正定义一个稍后不需要删除的事件处理程序，请使用 lambda 表达式。

```

  public Form2()
{
    // You can use a lambda expression to define an event handl
    this.Click += (s, e) =&gt;
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
```

```
// Using a lambda expression shortens the following tradition
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

静态成员

- 通过使用类名称调用静态成员：*ClassName.StaticMember*。这种做法通过明确静态访问使代码更易于阅读。请勿使用派生类的名称限定基类中定义的静态成员。编译该代码时，代码可读性具有误导性，如果向派生类添加具有相同名称的静态成员，代码可能会被破坏。

LINQ 查询

- 对查询变量使用有意义的名称。下面的示例为位于西雅图的客户使用 seattleCustomers。

```
var seattleCustomers = from cust in customers
                       where cust.City == "Seattle"
                       select cust.Name;
```

- 使用别名确保匿名类型的属性名称都使用 Pascal 大小写格式正确大写。

```
var localDistributors =
    from customer in customers
    join distributor in distributors on customer.City equals di
    select new { Customer = customer, Distributor = distributor
```

- 如果结果中的属性名称模棱两可，请对属性重命名。例如，如果你的查询返回客户名称和分销商 ID，而不是在结果中将它们保留为 Name 和 ID，请对它们进行重命名以明确 Name 是客户的名称，ID 是分销商的 ID。

```
var localDistributors2 =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorID = dist
```

- 在查询变量和范围变量的声明中使用隐式类型化。

```
var seattleCustomers = from cust in customers
                       where cust.City == "Seattle"
                       select cust.Name;
```

- 对齐 from 子句下的查询子句，如上面的示例所示。
- 在其他查询子句之前使用 where 子句，以确保后面的查询子句作用于经过减少和筛选的数据集。

```
var seattleCustomers2 = from cust in customers
                        where cust.City == "Seattle"
                        orderby cust.Name
                        select cust;
```

- 使用多行 **from** 子句代替 join 子句以访问内部集合。例如，Student 对象的集合可能包含测验分数的集合。当执行以下查询时，它返回高于 90 的分数，并返回得到该分数的学生的姓氏。

```
// Use a compound from to access the inner sequence within each
var scoreQuery = from student in students
                 from score in student.Scores
                 where score &gt; 90
                 select new { Last = student.LastName, score };
```

安全性

请遵循代码安全维护指南中的准则。

请参阅

Visual Basic 编码约定

代码安全维护指南

数组（C# 编程指南）

可以在一个数组数据结构中存储同一类型的多个变量。通过指定其元素的类型声明数组。

type[] arrayName;

下面的示例创建一维、多维和交错数组：

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array
        int[] array1 = new int[5];

        // Declare and set array element values
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 }

        // Declare a jagged array
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array st
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

数组概述

数组具有以下属性：

- 数组可以是一维、多维或交错的。
- 当创建了数组实例时，将建立维度数和每个维度的长度。在实例的生存期内，这些值不能更改。
- 数值数组元素的默认值设置为零，而引用元素的默认值设置为 null。

- 交错数组是数组的数组，因此其元素是引用类型并初始化为 **null**。
- 数组的索引从零开始：具有 n 个元素的数组的索引是从 0 到 n-1。
- 数组元素可以是任何类型，包括数组类型。
- 数组类型是从抽象基类型 Array 派生的引用类型。由于此类型实现了 IEnumerable 和 IEnumerable<T>，因此可以对 C# 中的所有数组使用 foreach 迭代。

相关章节

- 作为对象的数组（C# 编程指南）
- 对数组使用 foreach（C# 编程指南）
- 将数组作为参数传递（C# 编程指南）
- 使用 ref 和 out 传递数组（C# 编程指南）
- More About Variables（有关变量的更多信息）位于Beginning Visual C# 2010（开始 Visual C# 2010）中

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

集合（C# 和 Visual Basic）

Array Collection Type

作为对象的数组（C# 编程指南）

在 C# 中，数组实际上是对象，而不只是像 C 和 C++ 中那样的可寻址连续内存区域。 Array 是所有数组类型的抽象基类型。 可以使用 Array 具有的属性以及其他类成员。 这种用法的一个示例是使用 Length 属性来获取数组的长度。 下面的代码将 numbers 数组的长度（为 5）赋给名为 lengthOfNumbers 的变量：

```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
```

Array 类提供了许多其他有用的方法和属性，用于排序、搜索和复制数组。

示例

此示例使用 Rank 属性来显示数组的维数。

```
class TestArraysClass
{
    static void Main()
    {
        // Declare and initialize an array:
        int[,] theArray = new int[5, 10];
        System.Console.WriteLine("The array has {0} dimensions.", t
    }
}
// Output: The array has 2 dimensions.
```

请参见

参考

数组（C# 编程指南） 一维数组（C# 编程指南） 多维数组（C# 编程指南） 交错数组（C# 编程指南）

概念

C# 编程指南

一维数组（C# 编程指南）

可按下面的示例所示声明五个整数的一维数组。

```
int[] array = new int[5];
```

此数组包含从 array[0] 到 array[4] 的元素。 new 运算符用于创建数组并将数组元素初始化为它们的默认值。在此例中，所有数组元素都初始化为零。

可以用相同的方式声明存储字符串元素的数组。例如：

```
string[] stringArray = new string[6];
```

数组初始化

可以在声明数组时将其初始化，在这种情况下不需要级别说明符，因为级别说明符已经由初始化列表中的元素数提供。例如：

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

可以用相同的方式初始化字符串数组。下面声明一个字符串数组，其中每个数组元素用每天的名称初始化：

```
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sa
```

如果在声明数组时将其初始化，则可以使用下列快捷方式：

```
int[] array2 = { 1, 3, 5, 7, 9 };
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "S
```

可以声明一个数组变量但不将其初始化，但在将数组分配给此变量时必须使用 **new** 运算符。例如：

```
int[] array3;
array3 = new int[] { 1, 3, 5, 7, 9 };   // OK
//array3 = {1, 3, 5, 7, 9};   // Error
```

C# 3.0 引入了隐式类型的数组。有关更多信息，请参见 隐式类型的数组（C# 编程指南）。

值类型数组和引用类型数组

请看下列数组声明：

```
SomeType[] array4 = new SomeType[10];
```

该语句的结果取决于 SomeType 是值类型还是引用类型。如果为值类型，则语句将创建一个有 10 个元素的数组，每个元素都有 SomeType 类型。如果 SomeType 是引用类型，则该语句将创建一个由 10 个元素组成的数组，其中每个元素都初始化为空引用。

有关值类型和引用类型的更多信息，请参见类型（C# 参考）。

请参阅

Array

C# 编程指南

数组（C# 编程指南）

多维数组（C# 编程指南）

交错数组（C# 编程指南）

多维数组（C# 编程指南）

数组可以具有多个维度。例如，下列声明创建一个四行两列的二维数组。

```
int[,] array = new int[4, 2];
```

下列声明创建一个三维（4、2 和 3）数组。

```
int[, ,] array1 = new int[4, 2, 3];
```

数组初始化

可以在声明数组时将其初始化，如下例所示。

```
// Two-dimensional array.
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8
// The same array with dimensions specified.
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7
// A similar array with string elements.
string[,] array2Db = new string[3, 2] { { "one", "two" }, { "three"
                                        { "five", "six" } };

// Three-dimensional array.
int[, ,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                 { { 7, 8, 9 }, { 10, 11, 12 } } };
// The same array with dimensions specified.
int[, ,] array3Da = new int[2, 2, 3] { { { 1, 2, 3 }, { 4, 5, 6 } 
                                       { { 7, 8, 9 }, { 10, 11, 12

// Accessing array elements.
System.Console.WriteLine(array2D[0, 0]);
System.Console.WriteLine(array2D[0, 1]);
System.Console.WriteLine(array2D[1, 0]);
System.Console.WriteLine(array2D[1, 1]);
System.Console.WriteLine(array2D[3, 0]);
System.Console.WriteLine(array2Db[1, 0]);
System.Console.WriteLine(array3Da[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);

// Getting the total count of elements or the length of a given dir
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++) {
    total *= array3D.GetLength(i);
}
System.Console.WriteLine("{0} equals {1}", allLength, total);

// Output:
// 1
// 2
// 3
// 4
// 7
// three
// 8
// 12
// 12 equals 12
```

也可以初始化数组但不指定级别。

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

如果选择声明一个数组变量但不将其初始化，必须使用 **new** 运算符将一个数组分配给此变量。以下示例显示 **new** 的用法。

```
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };   /
//array5 = {{1,2}, {3,4}, {5,6}, {7,8}};   // Error
```

以下示例将值分配给特定的数组元素。

```
array5[2, 1] = 25;
```

同样，下面的示例获取特定数组元素的值，并将它赋给变量elementValue。

```
int elementValue = array5[2, 1];
```

以下代码示例将数组元素初始化为默认值（交错数组除外）：

```
int[,] array6 = new int[10, 10];
```

请参阅

C# 编程指南

数组（C# 编程指南）

一维数组（C# 编程指南）

交错数组（C# 编程指南）

交错数组（C# 编程指南）

交错数组是元素为数组的数组。交错数组元素的维度和大小可以不同。交错数组有时称为“数组的数组”。以下示例说明如何声明、初始化和访问交错数组。

下面声明一个由三个元素组成的一维数组，其中每个元素都是一个一维整数数组：

```
int[][] jaggedArray = new int[3][];
```

必须初始化 jaggedArray 的元素后才可以使用它。可以如下例所示初始化该元素：

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

每个元素都是一个一维整数数组。第一个元素是由 5 个整数组成的数组，第二个是由 4 个整数组成的数组，而第三个是由 2 个整数组成的数组。

也可以使用初始值设定项用值填充数组元素，在这种情况下不需要数组大小。例如：

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

还可以在声明数组时将其初始化，如：

```
    int[][] jaggedArray2 = new int[][]
{
    new int[] {1,3,5,7,9},
    new int[] {0,2,4,6},
    new int[] {11,22}
};
```

可以使用下面的速记格式。请注意：不能从元素初始化中省略 **new** 运算符，因为不存在元素的默认初始化：

```
    int[][] jaggedArray3 =
{
    new int[] {1,3,5,7,9},
    new int[] {0,2,4,6},
    new int[] {11,22}
};
```

交错数组是数组的数组，因此其元素是引用类型并初始化为 **null**。

可以如下例所示访问个别数组元素：

```
// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray3[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray3[2][1] = 88;
```

可以混合使用交错数组和多维数组。下面声明和初始化一个一维交错数组，该数组包含大小不同的三个二维数组元素。有关二维数组的详细信息，请参阅多维数组（C# 编程指南）。

```
int[][,] jaggedArray4 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};
```

可以如本例所示访问个别元素，该示例显示第一个数组的元素 [1,0] 的值（值为 5）：

```
System.Console.Write("{0}", jaggedArray4[0][1, 0]);
```

方法 Length 返回包含在交错数组中的数组的数目。例如，假定您已声明了前一个数组，则此行：

```
System.Console.WriteLine(jaggedArray4.Length);
```

返回值 3。

本例生成一个数组，该数组的元素为数组自身。每一个数组元素都有不同的大小。

```
class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements:
        int[][] arr = new int[2][];

        // Initialize the elements:
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Display the array elements:
        for (int i = 0; i < arr.Length; i++)
        {
            System.Console.Write("Element({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write("{0}{1}", arr[i][j], j == (ar
            }
            System.Console.WriteLine();
        }
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

请参阅

Array

C# 编程指南

数组（C# 编程指南）

一维数组（C# 编程指南）

多维数组（C# 编程指南）

对数组使用 foreach（C# 编程指南）

C# 还提供 foreach 语句。 该语句提供一种简单、明了的方法来循环访问数组或任何可枚举集合的元素。 foreach 语句按数组或集合类型的枚举器返回的顺序处理元素，该顺序通常是从第 0 个元素到最后一个元素。 例如，以下代码创建一个名为 numbers 的数组，并使用 foreach 语句循环访问该数组：

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}
// Output: 4 5 6 1 2 3 -2 -1 0
```

借助多维数组，你可以使用相同的方法来循环访问元素，例如：

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 }
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

但对于多维数组，使用嵌套的 for 循环可以更好地控制数组元素。

请参见

参考

数组（C# 编程指南） 一维数组（C# 编程指南） 多维数组（C# 编程指南） 交错数组（C# 编程指南） Array

概念

C# 编程指南

将数组作为参数传递（C# 编程指南）

数组可作为实参传递给方法形参。由于数组是引用类型，因此方法可以更改元素的值。

将一维数组作为参数传递

可以将初始化的一维数组传递给方法。例如，下面的语句将数组发送到 print 方法。

```
int[] theArray = { 1, 3, 5, 7, 9 };
PrintArray(theArray);
```

下面的代码显示 print 方法的部分实现。

```
void PrintArray(int[] arr)
{
    // Method code.
}
```

您可以在一个步骤中初始化和传递新数组，如下面的示例所示。

```
PrintArray(new int[] { 1, 3, 5, 7, 9 });
```

示例

说明

在下面的示例中，将初始化一个字符串数组并将其作为参数传递到字符串的 PrintArray 方法。该方法显示数组的元素。接下来，调用 ChangeArray 和 ChangeArrayElement 方法以演示通过值发送数组参数时不会阻止更改这些数组元素。

代码

```
class ArrayClass
{
    static void PrintArray(string[] arr)
    {
```

```
        for (int i = 0; i < arr.Length; i++)
        {
            System.Console.Write(arr[i] + "{0}", i < arr.Length - 1
        }
        System.Console.WriteLine();
    }

    static void ChangeArray(string[] arr)
    {
        // The following attempt to reverse the array does not pers
        // the method returns, because arr is a value parameter.
        arr = (arr.Reverse()).ToArray();
        // The following statement displays Sat as the first elemen
        System.Console.WriteLine("arr[0] is {0} in ChangeArray.", a
    }

    static void ChangeArrayElements(string[] arr)
    {
        // The following assignments change the value of individual
        // elements.
        arr[0] = "Sat";
        arr[1] = "Fri";
        arr[2] = "Thu";
        // The following statement again displays Sat as the first
        // in the array arr, inside the called method.
        System.Console.WriteLine("arr[0] is {0} in ChangeArrayEleme
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "F

        // Pass the array as an argument to PrintArray.
        PrintArray(weekDays);

        // ChangeArray tries to change the array by assigning somet
        // to the array in the method.
        ChangeArray(weekDays);

        // Print the array again, to verify that it has not been ch
        System.Console.WriteLine("Array weekDays after the call to
        PrintArray(weekDays);
        System.Console.WriteLine();

        // ChangeArrayElements assigns new values to individual arr
        // elements.
        ChangeArrayElements(weekDays);

        // The changes to individual elements persist after the met
        // Print the array, to verify that it has been changed.
        System.Console.WriteLine("Array weekDays after the call to
        PrintArray(weekDays);
```

```
    }
}
// Output:
// Sun Mon Tue Wed Thu Fri Sat
// arr[0] is Sat in ChangeArray.
// Array weekDays after the call to ChangeArray:
// Sun Mon Tue Wed Thu Fri Sat
//
// arr[0] is Sat in ChangeArrayElements.
// Array weekDays after the call to ChangeArrayElements:
// Sat Fri Thu Wed Thu Fri Sat
```

将多维数组作为参数传递

可采用与传递一维数组相同的方式将初始化的多维数组传递给方法。

```
int[,] theArray = { { 1, 2 }, { 2, 3 }, { 3, 4 } };
Print2DArray(theArray);
```

下面的代码显示 print 方法的部分声明，该方法接受一个二维数组作为其参数。

```
void Print2DArray(int[,] arr)
{
    // Method code.
}
```

您可以在一个步骤中初始化和传递新数组，如下面的示例所示。

```
Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } })
```

示例

说明

在下面的示例中，将初始化一个二维整数数组并将其传递到 Print2DArray 方法。该方法显示数组的元素。

代码

```
class ArrayClass2D
{
    static void Print2DArray(int[,] arr)
    {
        // Display the array elements.
        for (int i = 0; i < arr.GetLength(0); i++)
        {
            for (int j = 0; j < arr.GetLength(1); j++)
            {
                System.Console.WriteLine("Element({0},{1})={2}", i,
            }
        }
    }
    static void Main()
    {
        // Pass the array as an argument.
        Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
    /* Output:
        Element(0,0)=1
        Element(0,1)=2
        Element(1,0)=3
        Element(1,1)=4
        Element(2,0)=5
        Element(2,1)=6
        Element(3,0)=7
        Element(3,1)=8
    */
```

请参阅

C# 编程指南

数组（C# 编程指南）

一维数组（C# 编程指南）

多维数组（C# 编程指南）

交错数组（C# 编程指南）

使用 ref 和 out 传递数组（C# 编程指南）

与所有 out 参数一样，在使用数组类型的 **out** 参数前必须先为其赋值；即必须由被调用方为其赋值。例如：

```
static void TestMethod1(out int[] arr)
{
    arr = new int[10];   // definite assignment of arr
}
```

与所有 ref 参数一样，数组类型的 **ref** 参数必须由调用方明确赋值。因此，不需要由被调用方明确赋值。可以将数组类型的 **ref** 参数更改为调用的结果。例如，可以为数组赋以 null 值，或将其初始化为另一个数组。例如：

```
static void TestMethod2(ref int[] arr)
{
    arr = new int[10];   // arr initialized to a different array
}
```

下面两个示例演示了 **out** 与 **ref** 在将数组传递给方法时的用法差异。

在此示例中，在调用方（Main 方法）中声明数组 theArray，并在 FillArray 方法中初始化此数组。然后，数组元素将返回调用方并显示。

```
class TestOut
{
    static void FillArray(out int[] arr)
    {
        // Initialize the array:
        arr = new int[5] { 1, 2, 3, 4, 5 };
    }

    static void Main()
    {
        int[] theArray; // Initialization is not required

        // Pass the array to the callee using out:
        FillArray(out theArray);

        // Display the array elements:
        System.Console.WriteLine("Array elements are:");
        for (int i = 0; i < theArray.Length; i++)
        {
            System.Console.Write(theArray[i] + " ");
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
    /* Output:
        Array elements are:
        1 2 3 4 5
    */
```

在此示例中，在调用方（Main 方法）中初始化数组 theArray，并通过使用 **ref** 参数将其传递给 FillArray 方法。在 FillArray 方法中更新某些数组元素。然后，数组元素将返回调用方并显示。

```
class TestRef
{
    static void FillArray(ref int[] arr)
    {
        // Create the array on demand:
        if (arr == null)
        {
            arr = new int[10];
        }
        // Fill the array:
        arr[0] = 1111;
        arr[4] = 5555;
    }

    static void Main()
    {
        // Initialize the array:
        int[] theArray = { 1, 2, 3, 4, 5 };

        // Pass the array using ref:
        FillArray(ref theArray);

        // Display the updated array:
        System.Console.WriteLine("Array elements are:");
        for (int i = 0; i < theArray.Length; i++)
        {
            System.Console.Write(theArray[i] + " ");
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
    /* Output:
        Array elements are:
        1111 2 3 4 5555
    */
```

请参阅

ref（C# 参考）

out 参数修饰符（C# 参考）

C# 编程指南

数组（C# 编程指南）

一维数组（C# 编程指南）

多维数组（C# 编程指南）

交错数组（C# 编程指南）

隐式类型的数组（C# 编程指南）

可以创建隐式类型的数组，在这样的数组中，数组实例的类型是从数组初始值设定项中指定的元素推断而来的。有关任何隐式类型变量的规则也适用于隐式类型的数组。有关更多信息，请参见隐式类型的局部变量（C# 编程指南）。

在查询表达式中，隐式类型的数组通常与匿名类型以及对象初始值设定项和集合初始值设定项一起使用。

下面的示例演示如何创建隐式类型的数组：

```
class ImplicitlyTypedArraySample
{
    static void Main()
    {
        var a = new[] { 1, 10, 100, 1000 }; // int[]
        var b = new[] { "hello", null, "world" }; // string[]

        // single-dimension jagged array
        var c = new[]
{
    new[]{1,2,3,4},
    new[]{5,6,7,8}
};

        // jagged array of strings
        var d = new[]
{
    new[]{"Luca", "Mads", "Luke", "Dinesh"},
    new[]{"Karen", "Suma", "Frances"}
};
    }
}
```

请注意，在上一个示例中，没有在初始化语句的左侧对隐式类型的数组使用方括号。另请注意，交错数组就像一维数组那样使用 **new []** 进行初始化。

对象初始值设定项中的隐式类型的数组

创建包含数组的匿名类型时，必须在该类型的对象初始值设定项中对数组进行隐式类型化。在下面的示例中，contacts 是一个隐式类型的匿名类型数组，其中每个匿名类型都包含一个名为 PhoneNumbers 的数组。请注意，对象初始值设定项内部未使用 **var** 关键字。

```
        var contacts = new[]
{
    new {
            Name = " Eugene Zabokritski",
            PhoneNumbers = new[] { "206-555-0108", "425-555-0001" ]
        },
    new {
            Name = " Hanying Feng",
            PhoneNumbers = new[] { "650-555-0199" }
        }
};
```

请参阅

C# 编程指南

隐式类型的局部变量（C# 编程指南）

数组（C# 编程指南）

匿名类型（C# 编程指南）

对象和集合初始值设定项（C# 编程指南）

var（C# 参考）

LINQ 查询表达式（C# 编程指南）

类和结构（C# 编程指南）

类和结构是 .NET Framework 中的常规类型系统的两种基本构造。两者在本质上都属于数据结构，封装着一组整体作为一个逻辑单位的数据和行为。数据和行为是该类或结构的“成员”，它们包含各自的方法、属性和事件等（本主题后面列出了这些内容）。

类或结构的声明类似于蓝图，用于在运行时创建实例或对象。如果定义一个名为 Person 的类或结构，则 Person 为类型名称。如果声明并初始化 Person 类型的变量 p，则 p 称为 Person 的对象或实例。可以创建同一 Person 类型的多个实例，每个实例在其属性和字段中具有不同的值。

类是一种“引用类型”。创建类的对象时，对象赋值到的变量只保存对该内存的引用。将对象引用赋给新变量时，新变量引用的是原始对象。通过一个变量做出的更改将反映在另一个变量中，因为两者引用同一数据。

结构是一种值类型。创建结构时，结构赋值到的变量保存该结构的实际数据。将结构赋给新变量时，将复制该结构。因此，新变量和原始变量包含同一数据的两个不同的副本。对一个副本的更改不影响另一个副本。

类通常用于对较为复杂的行为建模，或对要在创建类对象后进行修改的数据建模。结构最适合一些小型数据结构，这些数据结构包含的数据以创建结构后不修改的数据为主。

有关更多信息，请参见类（C# 编程指南）、对象（C# 编程指南）和结构（C# 编程指南）。

下面的示例在 ProgrammingGuide 命名空间的顶级使用三个成员定义了 MyCustomClass。在 Program 类的 Main 方法中创建了 MyCustomClass 的一个实例（对象），并使用点表示法访问该对象的方法和属性。

```
namespace ProgrammingGuide
{
    // Class definition.
    public class MyCustomClass
    {
        // Class members:
        // Property.
        public int Number { get; set; }

        // Method.
        public int Multiply(int num)
        {
            return num * Number;
        }

        // Instance Constructor.
        public MyCustomClass()
        {
            Number = 0;
        }
    }
    // Another class definition. This one contains
    // the Main method, the entry point for the program.
    class Program
    {
        static void Main(string[] args)
        {
            // Create an object of type MyCustomClass.
            MyCustomClass myClass = new MyCustomClass();

            // Set the value of a public property.
            myClass.Number = 27;

            // Call a public method.
            int result = myClass.Multiply(4);
        }
    }
}
```

封装

“封装”有时被称为面向对象的编程的第一个支柱或原则。根据封装的原则，类或结构可以指定其每个成员对于该类或结构外部的代码的可访问性。可将无意在类或程序集外部使用的方法和变量隐藏起来，以减小编码错误或遭恶意利用的可能性。

有关类的更多信息，请参见类（C# 编程指南）和对象（C# 编程指南）。

成员

所有方法、字段、常量、属性和事件都必须在类型内部进行声明；这些称为类型的“成员”。与其他一些语言不同的是，C# 中没有全局变量或方法。即使是作为程序入口点的 Main 方法也必须在类或结构内部进行声明。下表列出了可在类或结构中声明的所有不同种类的成员。

- 字段
- 常量
- 属性
- 方法
- 构造函数
- 析构函数
- 事件
- 索引器
- 运算符
- 嵌套类型

辅助功能

有些方法和属性要供类或结构外部的代码（称为“客户端代码”）调用或访问。另有一些方法和属性可能仅供类或结构在自身内部使用。应限制您的代码的可访问性，只允许应当访问它们的客户端代码进行访问，这一点十分重要。使用访问修饰符 public、protected、internal、**protected internal** 和 private 可以指定类型及其成员对于客户端代码的可访问性。默认可访问性为 **private**。有关详细信息，请参阅 访问修饰符（C# 编程指南）。

Inheritance

类（而非结构）支持继承的概念。派生自另一个类（“基类”）的类将自动包含基类除构造函数和析构函数之外的所有公共、受保护和内部成员。有关更多信息，请参见继承（C# 编程指南）和多态性（C# 编程指南）。

可以将类声明为抽象类，表示该类的一个或多个方法不具有实现。抽象类虽然无法直接实例化，但可以用作其他类的基类，由其他类提供缺少的实现。还可以将类声明为密封类，以禁止其他类从该类继承。有关详细信息，请参阅 抽象类、密封类及类成员（C# 编程指南）。

接口

类和结构可以继承多个接口。从接口继承意味着该类型要实现该接口中定义的所有方法。有关详细信息，请参阅 接口（C# 编程指南）。

泛型类型

可以使用一个或多个类型参数来定义类和结构。客户端代码在创建类型的实例时提供类型。例如，System.Collections.Generic 命名空间中的 List<T> 类使用一个类型参数进行定义。客户端代码创建 List<string> 或 List<int> 的实例来指定列表中将包含的类型。有关详细信息，请参阅 泛型（C# 编程指南）。

静态类型

可以将类（不是结构）声明为静态。静态类只能包含静态成员，不能使用 new 关键字进行实例化。在程序加载时，静态类的一个副本将加载到内存中，可通过类名称访问该类的成员。类和结构都可以包含静态成员。有关详细信息，请参阅 静态类和静态类成员（C# 编程指南）。

嵌套类型

类或结构可以嵌套在其他类或结构中。有关更多信息，请参见嵌套类型。

分部类型

可以在一个代码文件中定义类、结构或方法的一部分，而在另一个代码文件中定义另一部分。有关更多信息，请参见分部类和方法。

对象初始值设定项

可以实例化和初始化类或结构对象以及对象的集合，无需显式调用其构造函数。有关更多信息，请参见对象和集合初始值设定项（C# 编程指南）。

匿名类型

在不方便或没必要创建命名类的情况下，例如当使用无需保留或传递给其他方法的数据结构填充列表时，可以使用匿名类型。有关更多信息，请参见匿名类型（C# 编程指南）。

扩展方法

通过创建一个单独的类型，然后将该类型的方法当作原始类型的方法来调用，可以在不必创建派生类的情况下对类进行“扩展”。有关更多信息，请参见扩展方法（C# 编程指南）。

隐式类型化局部变量

在类或结构方法中，可以使用隐式类型来指示编译器在编译时确定正确的类型。有关更多信息，请参见隐式类型的局部变量（C# 编程指南）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

类（C# 编程指南）

“类”是一种构造，通过使用该构造，您可以将其他类型的变量、方法和事件组合在一起，从而创建自己的自定义类型。类就像一个蓝图，它定义类型的数据和行为。如果类没有声明为静态类，客户端代码就可以创建赋给变量的“对象”或“实例”，从而使用该类。在对变量的所有引用都超出范围之前，该变量始终保持在内存中。所有引用都超出范围时，CLR 将标记该变量以供垃圾回收。如果类声明为静态类，则内存中只存在一个副本，并且客户端代码只能通过该类自身而不是“实例变量”访问该类。有关更多信息，请参见静态类和静态类成员（C# 编程指南）。

与结构不同，类支持“继承”，而继承是面向对象编程的基础特性。有关更多信息，请参见继承（C# 编程指南）。

声明类

类使用 class 关键字进行声明，如下面的示例所示：

```
public class Customer
{
    //Fields, properties, methods and events go here...
}
```

class 关键字前面是访问级别。由于在该例中使用 public，因此任何人都可以基于该类创建对象。类的名称位于 **class** 关键字的后面。定义的其余部分是类的主体，用于定义行为和数据。类的字段、属性、方法和事件统称为“类成员”。

创建对象

尽管有时类和对象可互换，但它们是不同的概念。类定义对象的类型，但它不是对象本身。对象是基于类的具体实体，有时称为类的实例。

通过使用 new 关键字（后跟对象将基于的类的名称）可以创建对象，如下所示：

```
Customer object1 = new Customer();
```

创建类的实例后，将向程序员传递回对该对象的引用。在前面的示例中，object1 是对基于 Customer 的对象的引用。此引用引用新对象，但不包含对象数据本身。实际上，可以在根本不创建对象的情况下创建对象引用：

```
Customer object2;
```

建议不要创建像这样的不引用对象的对象引用，因为在运行时通过这样的引用来访问对象的尝试将会失败。但是，可以创建这样的引用来引用对象，方法是创建新对象，或者将它分配给现有的对象，如下所示：

```
Customer object3 = new Customer();
Customer object4 = object3;
```

此代码创建了两个对象引用，它们引用同一个对象。因此，通过 object3 对对象所做的任何更改都将反映在随后使用的 object4 中。由于基于类的对象是按引用来引用的，因此类称为引用类型。

类继承

继承是通过使用“派生”来实现的，而派生意味着类是使用“基类”声明的，它的数据和行为从基类继承。通过在派生的类名后面追加冒号和基类名称，可以指定基类，如下所示：

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherite
    // New Manager fields, properties, methods and events go here..
}
```

当类声明基类时，它继承基类除构造函数以外的所有成员。

与 C++ 不同，C# 中的类只能直接从一个基类继承。但是，因为基类自身也可能继承自另一个类，所以类可以间接继承多个基类。而且，一个类可以直接实现一个以上的接口。有关更多信息，请参见接口（C# 编程指南）。

类可以声明为抽象类。抽象类包含具有签名定义但没有实现的抽象方法。抽象类不能进行实例化。只能通过实现抽象方法的派生类使用抽象类。相比之下，密封类不允许其他类从其派生。有关更多信息，请参见抽象类、密封类及类成员（C# 编程指南）。

类定义可在不同的源文件之间进行拆分。有关更多信息，请参见分部类和方法（C# 编程指南）。

描述

下面的示例中定义了一个公共类，其中包含一个字段、一个方法和一个称为构造函数的特殊方法。有关更多信息，请参见构造函数（C# 编程指南）。然后使用 **new** 关键字将该类实例化。

```
public class Person
{
    // Field
    public string name;

    // Constructor that takes no arguments.
    public Person()
    {
        name = "unknown";
    }

    // Constructor that takes one argument.
    public Person(string nm)
    {
        name = nm;
    }

    // Method
    public void SetName(string newName)
    {
        name = newName;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        Person person1 = new Person();
        Console.WriteLine(person1.name);

        person1.SetName("John Smith");
        Console.WriteLine(person1.name);

        // Call the constructor that has one parameter.
        Person person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.name);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// John Smith
// Sarah Jones
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

面向对象的编程（C# 和 Visual Basic）

多态性（C# 编程指南）

成员（C# 编程指南）

方法（C# 编程指南）

构造函数（C# 编程指南）

析构函数（C# 编程指南）

对象（C# 编程指南）

对象（C# 编程指南）

类或结构定义的作用类似于蓝图，指定该类型可以进行哪些操作。从本质上说，对象是按照此蓝图分配和配置的内存块。程序可以创建同一个类的多个对象。对象也称为实例，可以存储在命名变量中，也可以存储在数组或集合中。使用这些变量来调用对象方法及访问对象公共属性的代码称为客户端代码。在 C# 等面向对象的语言中，典型的程序由动态交互的多个对象组成。

注意
静态类型的行为与此处介绍的不同。有关更多信息，请参见静态类和静态类成员（C# 编程指南）。

结构实例。. 选件类实例

由于类是引用类型，因此类对象的变量引用该对象在托管堆上的地址。如果将同一类型的第二个对象分配给第一个对象，则两个变量都引用该地址的对象。这一点将在本主题后面部分进行更详细的讨论。

类的实例是使用 new 运算符创建的。在下面的示例中，Person 为类型，person1 和 person 2 为该类型的实例（即对象）。

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    //Other properties, methods, events...
}

class Program
{
    static void Main()
    {
        Person person1 = new Person("Leopold", 6);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.N

        // Declare  new person, assign person1 to it.
        Person person2 = person1;

        //Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine("person2 Name = {0} Age = {1}", person2.N
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.N

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();

    }
}
/*
    Output:
    person1 Name = Leopold Age = 6
    person2 Name = Molly Age = 16
    person1 Name = Molly Age = 16
*/
```

由于结构是值类型，因此结构对象的变量具有整个对象的副本。结构的实例也可以使用 **new** 运算符来创建，但这不是必需的，如下面的示例所示：

```
public struct Person
{
    public string Name;
    public int Age;
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

public class Application
{
    static void Main()
    {
        // Create  struct instance and initialize by using "new".
        // Memory is allocated on thread stack.
        Person p1 = new Person("Alex", 9);
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Ag

        // Create  new struct object. Note that  struct can be init
        // without using "new".
        Person p2 = p1;

        // Assign values to p2 members.
        p2.Name = "Spencer";
        p2.Age = 7;
        Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Ag

        // p1 values remain unchanged because p2 is  copy.
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Ag

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
  Output:
    p1 Name = Alex Age = 9
    p2 Name = Spencer Age = 7
    p1 Name = Alex Age = 9
*/
```

p1 和 p2 的内存在线程堆栈上进行分配。该内存随声明它的类型或方法一起回收。这就是在赋值时复制结构的一个原因。相比之下，当对类实例对象的所有引用都超出范围时，为该类实例分配的内存将由公共语言运行时自动回收（垃圾回收）。无法像在 C++ 中那样明确地销毁类对象。有关 .NET Framework 中的垃圾回收的更多信息，请参见Garbage Collection。

> 注意
>
> 公共语言运行时中高度优化了托管堆上内存的分配和释放。在大多数情况下，在堆上分配类实例与在堆栈上分配结构实例在性能开销上没有显著的差别。

对象标识与. 值相等性

在比较两个对象是否相等时，首先必须明确您是想知道两个变量是否表示内存中的同一对象，还是想知道这两个对象的一个或多个字段的值是否相等。如果您要对值进行比较，则必须考虑这两个对象是值类型（结构）的实例，还是引用类型（类、委托、数组）的实例。

- 若要确定两个类实例是否引用内存中的同一位置（意味着它们具有相同的标识），可使用静态 Equals 方法。（System.Object 是所有值类型和引用类型的隐式基类，其中包括用户定义的结构和类。）
- 若要确定两个结构实例中的实例字段是否具有相同的值，可使用 ValueType.Equals 方法。由于所有结构都隐式继承自 System.ValueType，因此可以直接在对象上调用该方法，如下面的示例所示：

```
// Person is defined in the previous example.

//public struct Person
//{
//    public string Name;
//    public int Age;
//    public Person(string name, int age)
//    {
//        Name = name;
//        Age = age;
//    }
//}

Person p1 = new Person("Wallace", 75);
Person p2;
p2.Name = "Wallace";
p2.Age = 75;

if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.
```

Equals 的 System.ValueType 实现使用反射，因为它必须能够确定任何结构中有哪些字段。在创建您自己的结构时，重写 Equals 方法可以提供针对您的类型的高效求等算法。

- 要确定两个类实例中字段的值是否相等，您可以使用 Equals 方法或 == 运算

符。但是，只有类通过已重写或重载提供关于那种类型对象的相等含义的自定义时，才能使用它们。 类也可以实现 IEquatable<T> 接口或 IEqualityComparer<T> 接口。 这两个接口都提供可用于测试值相等性的方法。设计好重写 **Equals** 的类后，请务必遵循如何：为类型定义值相等性（C# 编程指南）和 Object.Equals(Object) 中介绍的准则。

相关章节

有关更多信息：

- 类（C# 编程指南）
- 结构（C# 编程指南）
- 构造函数（C# 编程指南）
- 析构函数（C# 编程指南）
- 事件（C# 编程指南）

请参阅

C# 编程指南

object（C# 参考）

继承（C# 编程指南）

class（C# 参考）

struct（C# 参考）

new 运算符（C# 参考）

常规类型系统

结构（C# 编程指南）

结构是使用 struct 关键字定义的，例如：

```
public struct PostalAddress
{
    // Fields, properties, methods and events go here...
}
```

结构与类共享大多数相同的语法，但结构比类受到的限制更多：

- 在结构声明中，除非字段被声明为 const 或 static，否则无法初始化。
- 结构不能声明默认构造函数（没有参数的构造函数）或析构函数。
- 结构在赋值时进行复制。将结构赋值给新变量时，将复制所有数据，并且对新副本所做的任何修改不会更改原始副本的数据。在使用值类型的集合（如 Dictionary<string, myStruct>）时，请务必记住这一点。
- 结构是值类型，而类是引用类型。
- 与类不同，结构的实例化可以不使用 **new** 运算符。
- 结构可以声明带参数的构造函数。
- 一个结构不能从另一个结构或类继承，而且不能作为一个类的基。所有结构都直接继承自 **System.ValueType**，后者继承自 **System.Object**。
- 结构可以实现接口。
- 结构可用作可以为 null 的类型，因而可向其赋 null 值。

相关章节

有关更多信息：

- 使用结构（C# 编程指南）
- 构造函数（C# 编程指南）
- 可以为 null 的类型（C# 编程指南）
- 如何：了解向方法传递结构和向方法传递类引用之间的区别（C# 编程指南）
- 如何：在结构间实现用户定义的转换（C# 编程指南）
- More About Variables 在 Beginning Visual C# 2010

请参阅

C# 编程指南

类和结构（C# 编程指南）

类（C# 编程指南）

继承（C# 编程指南）

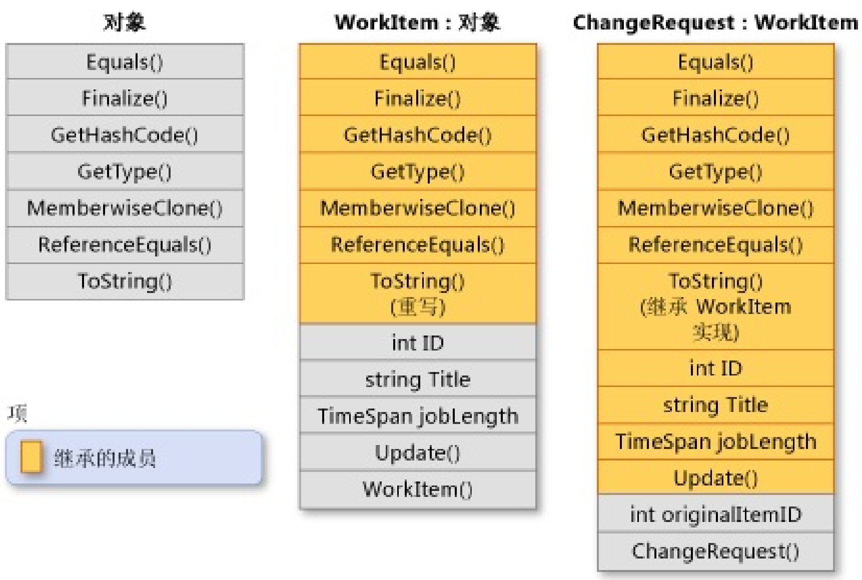
继承（加上封装和多态性）是面向对象的编程的三个主要特性（也称为“支柱”）之一。 继承用于创建可重用、扩展和修改在其他类中定义的行为的新类。其成员被继承的类称为“基类”，继承这些成员的类称为“派生类”。派生类只能有一个直接基类。但是，继承是可传递的。如果 ClassB 派生出 ClassC，ClassA 派生出 ClassB，则 ClassC 会继承 ClassB 和 ClassA 中声明的成员。

> 注意
>
> 结构不支持继承，但可以实现接口。有关更多信息，请参见 接口（C# 编程指南）。

从概念上来说，派生类是基类的特例。 例如，如果您有一个基类 Animal，则可以有一个名为 Mammal 的派生类和一个名为 Reptile 的派生类。 Mammal 是一个 Animal，Reptile 也是一个 Animal，但每个派生类均表示基类的不同专用化。

定义一个类从其他类派生时，派生类隐式获得基类的除构造函数和析构函数以外的所有成员。因此，派生类可以重用基类中的代码而无需重新实现这些代码。可以在派生类中添加更多成员。派生类以这种方式扩展基类的功能。

下图演示一个 WorkItem 类，该类表示某业务流程中的一个工作项。和所有的类一样，该类派生自 System.Object 并继承其所有方法。 WorkItem 添加了自己的五个成员。其中包括一个构造函数，因为构造函数不能继承。类ChangeRequest 继承自 WorkItem 并表示特定种类的工作项。 ChangeRequest 在它从 WorkItem 和 Object 继承的成员中另外添加了两个成员。它必须添加其自己的构造函数，还添加 originalItemID。利用属性 originalItemID，可将 ChangeRequest 实例与更改请求将应用到的原始 WorkItem 相关联。



类继承

下面的示例演示如何以 C# 表示上图所示的类关系。该示例还演示 WorkItem 如何重写虚方法 Object.ToString，以及 ChangeRequest 类如何继承该方法的 WorkItem 实现。

```
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkIte
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a ba
    // class constructor explicitly, the default constructor is cal
    // implicitly.
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentI
    // constructor is called one time, automatically, before any in
    // of WorkItem or ChangeRequest is created, or currentID is ref
    static WorkItem()
    {
        currentID = 0;
    }

    protected int GetNextID()
    {
        // currentID is a static field. It is incremented each time
        // instance of WorkItem is created.
        return ++currentID;
    }
```

```
    // Method Update enables you to update the title and job length
    // existing WorkItem object.
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inher
    // from System.Object.
    public override string ToString()
    {
        return String.Format("{0} - {1}", this.ID, this.Title);
    }
}

// ChangeRequest derives from WorkItem and adds a property (origina
// and two constructors.
public class ChangeRequest : WorkItem
{
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
    // constructor explicitly, the default constructor in the base
    // is called implicitly. The base class must contain a default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
    public ChangeRequest(string title, string desc, TimeSpan jobLen
                         int originalID)
    {
        // The following properties and the GetNexID method are inh
        // from WorkItem.
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = jobLen;

        // Property originalItemId is a member of ChangeRequest, bu
        // of WorkItem.
        this.originalItemID = originalID;
    }
}

class Program
{
    static void Main()
    {
        // Create an instance of WorkItem by using the constructor
        // base class that takes three arguments.
```

```
        WorkItem item = new WorkItem("Fix Bugs",
                                    "Fix all bugs in my code branc
                                    new TimeSpan(3, 4, 0, 0));

        // Create an instance of ChangeRequest by using the constru
        // the derived class that takes four arguments.
        ChangeRequest change = new ChangeRequest("Change Base Class
                                                "Add members to th
                                                new TimeSpan(4, 0,
                                                1);

        // Use the ToString method defined in WorkItem.
        Console.WriteLine(item.ToString());

        // Use the inherited Update method to change the title of t
        // ChangeRequest object.
        change.Update("Change the Design of the Base Class",
            new TimeSpan(4, 0, 0));

        // ChangeRequest inherits WorkItem's override of ToString.
        Console.WriteLine(change.ToString());

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    1 - Fix Bugs
    2 - Change the Design of the Base Class
*/
```

抽象方法和虚方法

当基类将方法声明为 virtual 时，派生类可以用自己的实现重写该方法。如果基类将成员声明为 abstract，则在直接继承自该类的任何非抽象类中都必须重写该方法。如果派生类自身是抽象的，则它继承抽象成员而不实现它们。抽象成员和虚成员是多态性的基础，多态性是面向对象的编程的第二个主要特性。有关更多信息，请参见 多态性（C# 编程指南）。

抽象基类

如果希望禁止通过 new关键字直接进行实例化，可以将类声明为abstract如果这样做，则仅当从该类派生新类时才能使用该类。抽象类可以包含一个或多个自身声明为抽象的方法签名。这些签名指定参数和返回值，但没有实现（方法体）。抽象类

不必包含抽象成员；但是，如果某个类确实包含抽象成员，则该类自身必须声明为抽象类。自身不是抽象类的派生类必须为抽象基类中的任何抽象方法提供实现。有关更多信息，请参见 抽象类、密封类及类成员（C# 编程指南）。

接口

“接口”是一种引用类型，有点像仅包含抽象成员的抽象基类。类在从接口实现时必须为该接口的所有成员提供实现。类虽然只能从一个直接基类派生，但可以实现多个接口。

接口用于为不一定具有“是”关系的类定义特定功能。例如，System.IEquatable<T> 接口可由任何类或构造实现，这些类或构造必须启用代码来确定该类型的两个对象是否等效（但是该类型定义等效性）。 IEquatable<T> 不表示基类和派生类之间存在的同一种“是”关系（例如 Mammal 是 Animal）。有关更多信息，请参见 接口（C# 编程指南）。

派生类对基类成员的访问

派生类可以访问基类的公共成员、受保护成员、内部成员和受保护内部成员。即使派生类继承基类的私有成员，仍不能访问这些成员。但是，所有这些私有成员在派生类中仍然存在，且执行与基类自身中相同的工作。例如，假定一个受保护基类方法访问私有字段。要使继承的基类方法正常工作，派生类中必须有该字段。

禁止进一步派生

类可以将自身或其成员声明为 sealed，从而禁止其他类从该类自身或其任何成员继承。有关更多信息，请参见 抽象类、密封类及类成员（C# 编程指南）。

派生类隐藏基类成员

派生类可以通过以相同的名称和签名声明基类成员来隐藏这些成员。可以使用 new 修饰符显式指示成员不作为基类成员的重写。不是必须要使用 new，但如果不使用 new，将生成编译器警告。有关更多信息，请参见使用 Override 和 New 关键字进行版本控制（C# 编程指南）和 了解何时使用 Override 和 New 关键字（C# 编程指南）。

请参阅

C# 编程指南

类和结构（C# 编程指南）

class（C# 参考）

struct（C# 参考）

多态性（C# 编程指南）

多态性常被视为自封装和继承之后，面向对象的编程的第三个支柱。Polymorphism（多态性）是一个希腊词，指"多种形态"，多态性具有两个截然不同的方面：

- 在运行时，在方法参数和集合或数组等位置，派生类的对象可以作为基类的对象处理。发生此情况时，该对象的声明类型不再与运行时类型相同。
- 基类可以定义并实现虚方法，派生类可以重写这些方法，即派生类提供自己的定义和实现。在运行时，客户端代码调用该方法，CLR 查找对象的运行时类型，并调用虚方法的重写方法。因此，你可以在源代码中调用基类的方法，但执行该方法的派生类版本。

虚方法允许你以统一方式处理多组相关的对象。例如，假定你有一个绘图应用程序，允许用户在绘图图面上创建各种形状。你在编译时不知道用户将创建哪些特定类型的形状。但应用程序必须跟踪创建的所有类型的形状，并且必须更新这些形状以响应用户鼠标操作。你可以使用多态性通过两个基本步骤解决这一问题：

1. 创建一个类层次结构，其中每个特定形状类均派生自一个公共基类。
2. 使用虚方法通过对基类方法的单个调用来调用任何派生类上的相应方法。

首先，创建一个名为 Shape 的基类，并创建一些派生类，例如 Rectangle、Circle 和 Triangle。为 Shape 类提供一个名为 Draw 的虚方法，并在每个派生类中重写该方法以绘制该类表示的特定形状。创建一个 List<Shape> 对象，并向该对象添加 Circle、Triangle 和 Rectangle。若要更新绘图图面，请使用 foreach 循环对该列表进行循环访问，并对其中的每个 Shape 对象调用 Draw 方法。虽然列表中的每个对象都具有声明类型 Shape，但调用的将是运行时类型（该方法在每个派生类中的重写版本）。

```
public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

class Circle : Shape
{
    public override void Draw()
```

```
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Polymorphism at work #1: a Rectangle, Triangle and Circl
        // can all be used whereever a Shape is expected. No cast i
        // required because an implicit conversion exists from a de
        // class to its base class.
        System.Collections.Generic.List<Shape> shapes = new System.
        shapes.Add(new Rectangle());
        shapes.Add(new Triangle());
        shapes.Add(new Circle());

        // Polymorphism at work #2: the virtual method Draw is
        // invoked on each of the derived classes, not the base cla
        foreach (Shape s in shapes)
        {
            s.Draw();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

}

/* Output:
```

```
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
*/
```

在 C# 中，每个类型都是多态的，因为包括用户定义类型在内的所有类型都继承自 Object。

多态性概述

虚成员

当派生类从基类继承时，它会获得基类的所有方法、字段、属性和事件。派生类的设计器可以选择是否

- 重写基类中的虚拟成员。
- 继承最接近的基类方法而不重写它
- 定义隐藏基类实现的成员的新非虚实现

仅当基类成员声明为 virtual 或 abstract 时，派生类才能重写基类成员。派生成员必须使用 override 关键字显式指示该方法将参与虚调用。以下代码提供了一个示例：

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

字段不能是虚拟的，只有方法、属性、事件和索引器才可以是虚拟的。当派生类重写某个虚拟成员时，即使该派生类的实例被当作基类的实例访问，也会调用该成员。以下代码提供了一个示例：

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Also calls the new method.
```

虚方法和属性允许派生类扩展基类，而无需使用方法的基类实现。有关详细信息，请参阅使用 Override 和 New 关键字进行版本控制（C# 编程指南）。接口提供另一种方式来定义将实现留给派生类的方法或方法集。有关详细信息，请参阅接口（C# 编程指南）。

使用新成员隐藏基类成员

如果希望派生成员具有与基类中的成员相同的名称，但又不希望派生成员参与虚调用，则可以使用 new 关键字。 **new** 关键字放置在要替换的类成员的返回类型之前。以下代码提供了一个示例：

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

通过将派生类的实例强制转换为基类的实例，仍然可以从客户端代码访问隐藏的基类成员。例如：

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

阻止派生类重写虚拟成员

无论在虚拟成员和最初声明虚拟成员的类之间已声明了多少个类，虚拟成员永远都是虚拟的。如果类 A 声明了一个虚拟成员，类 B 从 A 派生，类 C 从类 B 派生，则类 C 继承该虚拟成员，并且可以选择重写它，而不管类 B 是否为该成员声明了重写。以下代码提供了一个示例：

```
public class A
{
    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}
```

派生类可以通过将重写声明为 sealed 来停止虚拟继承。这需要在类成员声明中的 **override** 关键字前面放置 **sealed** 关键字。以下代码提供了一个示例：

```
public class C : B
{
    public sealed override void DoWork() { }
}
```

在上一个示例中，方法 DoWork 对从 C 派生的任何类都不再是虚拟方法。即使它们转换为类型 B 或类型 A，它对于 C 的实例仍然是虚拟的。通过使用 **new** 关键字，密封的方法可以由派生类替换，如下面的示例所示：

```
public class D : C
{
    public new void DoWork() { }
}
```

在此情况下，如果在 D 中使用类型为 D 的变量调用 DoWork，被调用的将是新的 DoWork。如果使用类型为 C、B 或 A 的变量访问 D 的实例，对 DoWork 的调用将遵循虚拟继承的规则，即把这些调用传送到类 C 的 DoWork 实现。

从派生类访问基类虚拟成员

已替换或重写某个方法或属性的派生类仍然可以使用基关键字访问基类的该方法或属性。以下代码提供了一个示例：

```
public class Base
{
    public virtual void DoWork() {/*...*/ }
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
        //...
        // Call DoWork on base class
        base.DoWork();
    }
}
```

有关详细信息，请参阅 base。

> 注意
>
> 建议虚拟成员在它们自己的实现中使用 **base** 来调用该成员的基类实现。允许基类行为发生使得派生类能够集中精力实现特定于派生类的行为。未调用基类实现时，由派生类负责使它们的行为与基类的行为兼容。

本节内容

- 使用 Override 和 New 关键字进行版本控制（C# 编程指南）
- 了解何时使用 Override 和 New 关键字（C# 编程指南）
- 如何：重写 ToString 方法（C# 编程指南）

请参阅

C# 编程指南

C# 编程指南

继承（C# 编程指南）

抽象类、密封类及类成员（C# 编程指南）

方法（C# 编程指南）

事件（C# 编程指南）

属性（C# 编程指南）

索引器（C# 编程指南）

类型（C# 编程指南）

抽象类、密封类及类成员（C# 编程指南）

使用 abstract 关键字可以创建不完整且必须在派生类中实现的类和类成员。

使用 sealed 关键字可以防止继承以前标记为 virtual 的类或某些类成员。

抽象类和类成员

通过在类定义前面放置关键字 **abstract**，可以将类声明为抽象类。例如：

```
public abstract class A
{
    // Class members here.
}
```

抽象类不能实例化。抽象类的用途是提供一个可供多个派生类共享的通用基类定义。例如，类库可以定义一个抽象类，将其用作多个类库函数的参数，并要求使用该库的程序员通过创建派生类来提供自己的类实现。

抽象类也可以定义抽象方法。方法是将关键字 **abstract** 添加到方法的返回类型的前面。例如：

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

抽象方法没有实现，所以方法定义后面是分号，而不是常规的方法块。抽象类的派生类必须实现所有抽象方法。当抽象类从基类继承虚方法时，抽象类可以使用抽象方法重写该虚方法。例如：

```
// compile with: /target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

如果将 **virtual** 方法声明为 **abstract**，则该方法对于从抽象类继承的所有类而言仍然是虚方法。继承一个抽象方法的类不能访问该方法的原始实现。在上一个示例中，类 F 中的 DoWork 不能调用类 D 中的 DoWork。通过这种方式，抽象类可以强制派生类为虚方法提供新的方法实现。

密封类和类成员

通过在类定义前面放置关键字 **sealed**，可以将类声明为密封类。例如：

```
public sealed class D
{
    // Class members here.
}
```

密封类不能用作基类。因此，它也不能是抽象类。密封类禁止派生。由于密封类从不用作基类，所以有些运行时优化可以略微提高密封类成员的调用速度。

在对基类的虚成员进行重写的派生类上，方法、索引器、属性或事件可以将该成员声明为密封成员。在用于以后的派生类时，这将取消成员的虚效果。方法是在类成员声明中将 **sealed** 关键字置于 override 关键字的前面。例如：

```
public class D : C
{
    public sealed override void DoWork() { }
}
```

请参阅

C# 编程指南

类和结构（C# 编程指南）

继承（C# 编程指南）

方法（C# 编程指南）

字段（C# 编程指南）

如何：定义抽象属性（C# 编程指南）

静态类和静态类成员（C# 编程指南）

静态类与非静态类基本相同，但存在一个区别：静态类不能实例化。也就是说，不能使用 new 关键字创建静态类类型的变量。因为没有实例变量，所以要使用类名本身访问静态类的成员。例如，如果名为 UtilityClass 的静态类有一个名为 MethodA 的公共方法，则按下面的示例所示调用该方法：

```
UtilityClass.MethodA();
```

对于只对输入参数进行运算而不获取或设置任何内部实例字段的方法集，静态类可以方便地用作这些方法集的容器。例如，在 .NET Framework 类库中，静态类 System.Math 包含的方法只执行数学运算，而无需存储或检索特定 Math 类实例特有的数据。就是说，通过指定类名称和方法名称来应用类成员，如下示例所述。

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

和所有类类型一样，当加载引用静态类的程序时，.NET Framework 公共语言运行时 (CLR) 将加载该静态类的类型信息。程序不能指定加载静态类的确切时间。但是，可以保证在程序中首次引用该类前加载该类，并初始化该类的字段并调用其静态构造函数。静态构造函数仅调用一次，在程序驻留的应用程序域的生存期内，静态类一直保留在内存中。

> 注意
>
> 若要创建仅允许创建一个自身实例的非静态类，请参见 Implementing Singleton in C#（在 C# 中实现单一实例）。

下表介绍静态类的主要特性：

- 仅包含静态成员。
- 无法实例化。
- 是密封的。
- 不能包含实例构造函数。

因此，创建静态类与创建仅包含静态成员和私有构造函数的类基本相同。私有构造函数阻止类被实例化。使用静态类的优点在于，编译器能够执行检查以确保不致偶然地添加实例成员。编译器将保证不会创建此类的实例。

静态类是密封的，因此不可被继承。它们不能从除 Object 外的任何类中继承。静态类不能包含实例构造函数，但可以包含静态构造函数。如果非静态类包含需要进行重要的初始化的静态成员，也应定义静态构造函数。有关更多信息，请参见 静态构造函数（C# 编程指南）。

下面是一个静态类的示例，它包含两个在摄氏温度和华氏温度之间执行来回转换的方法：

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCels
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahr
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1\. From Celsius to Fahrenheit.");
        Console.WriteLine("2\. From Fahrenheit to Celsius.");
        Console.Write(":");

        string selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
```

```
                Console.Write("Please enter the Celsius temperature
                F = TemperatureConverter.CelsiusToFahrenheit(Consol
                Console.WriteLine("Temperature in Fahrenheit: {0:F2
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperat
                C = TemperatureConverter.FahrenheitToCelsius(Consol
                Console.WriteLine("Temperature in Celsius: {0:F2}",
                break;

            default:
                Console.WriteLine("Please select a convertor.");
                break;
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Example Output:
    Please select the convertor direction
    1\. From Celsius to Fahrenheit.
    2\. From Fahrenheit to Celsius.
    :2
    Please enter the Fahrenheit temperature: 20
    Temperature in Celsius: -6.67
    Press any key to exit.
 */
```

静态成员

非静态类可以包含静态的方法、字段、属性或事件。即使没有创建类的实例，也可以调用该类中的静态成员。始终通过类名而不是实例名称访问静态成员。无论对一个类创建多少个实例，它的静态成员都只有一个副本。静态方法和属性不能访问其包含类型中的非静态字段和事件，并且不能访问任何对象的实例变量（除非在方法参数中显式传递）。

更常见的做法是声明具有一些静态成员的非静态类，而不是将整个类声明为静态类。静态字段有两个常见的用法：一是记录已实例化对象的个数，二是存储必须在所有实例之间共享的值。

静态方法可以被重载但不能被重写，因为它们属于类，不属于类的任何实例。

虽然字段不能声明为 static const，但 const 字段的行为在本质上是静态的。这样的字段属于类型，不属于类型的实例。因此，可以同对待静态字段一样使用 ClassName.MemberName 表示法来访问 const 字段。不需要对象实例。

C# 不支持静态局部变量（在方法范围内声明的变量）。

通过在成员的返回类型之前使用 **static** 关键字可以声明静态类成员，如下面的示例所示：

```
public class Automobile
{
    public static int NumberOfWheels = 4;
    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }
    public static void Drive() { }
    public static event EventType RunOutOfGas;

    // Other non-static fields and properties...
}
```

静态成员在第一次被访问之前并且在调用静态构造函数（如有存在）之前进行初始化。若要访问静态类成员，应使用类名而不是变量名来指定该成员的位置，如下面的示例所示：

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

如果类包含静态字段，请提供在加载类时初始化这些字段的静态构造函数。

对静态方法的调用以 Microsoft 中间语言 (MSIL) 生成调用指令，而对实例方法的调用生成 **callvirt** 指令，该指令还检查 null 对象引用。但是，两者之间的性能差异在大多数时候并不明显。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

static（C# 参考）

类（C# 编程指南）

class（C# 参考）

静态构造函数（C# 编程指南）

实例构造函数（C# 编程指南）

成员（C# 编程指南）

类和结构具有表示其数据和行为的成员。类的成员包括在类中声明的所有成员，以及在该类的继承层次结构中的所有类中声明的所有成员（构造函数和析构函数除外）。基类中的私有成员被继承，但不能从派生类访问。

下表列出类或结构中可包含的成员类型：

成员	描述
字段（C# 编程指南）	字段是在类范围声明的变量。字段可以是内置数值类型或其他类的实例。例如，日历类可能具有一个包含当前日期的字段。
常量（C# 编程指南）	常量是在编译时设置其值并且不能更改其值的字段或属性。
属性（C# 编程指南）	属性是类中可以像类中的字段一样访问的方法。属性可以为类字段提供保护，以避免字段在对象不知道的情况下被更改。
方法（C# 编程指南）	方法定义类可以执行的操作。方法可以接受提供输入数据的参数，并且可以通过参数返回输出数据。方法还可以不使用参数而直接返回值。
事件（C# 编程指南）	事件向其他对象提供有关发生的事情（如单击按钮或成功完成某个方法）的通知。事件是使用委托定义和触发的。
运算符（C# 编程指南）	重载运算符被视为类成员。在重载运算符时，在类中将该运算符定义为公共静态方法。预定义运算符（+、*、< 等）不考虑作为成员。有关详细信息，请参阅可重载运算符（C# 编程指南）。
索引器（C# 编程指南）	使用索引器可以用类似于数组的方式为对象建立索引。
构造函数（C# 编程指南）	构造函数是在第一次创建对象时调用的方法。它们通常用于初始化对象的数据。
析构函数（C# 编程指南）	C# 中极少使用析构函数。析构函数是当对象即将从内存中移除时由运行时执行引擎调用的方法。它们通常用来确保任何必须释放的资源都得到适当的处理。
嵌套类型（C# 编程指南）	嵌套类型是在其他类型中声明的类型，通常用于描述仅由包含它们的类型所使用的对象。

请参阅

C# 编程指南

类（C# 编程指南）

方法（C# 编程指南）

构造函数（C# 编程指南）

析构函数（C# 编程指南）

属性（C# 编程指南）

字段（C# 编程指南）

索引器（C# 编程指南）

事件（C# 编程指南）

嵌套类型（C# 编程指南）

运算符（C# 编程指南）

可重载运算符（C# 编程指南）

访问修饰符（C# 编程指南）

所有类型和类型成员都具有可访问性级别，用来控制是否可以在您程序集的其他代码中或其他程序集中使用它们。可使用一下访问修饰符指定声明类型或成员时类型或成员的可访问性。

public

同一程序集中的任何其他代码或引用该程序集的其他程序集都可以访问该类型或成员。

private

只有同一类或结构中的代码可以访问该类型或成员。

protected

只有同一类或结构或者此类的派生类中的代码才可以访问的类型或成员。

internal

同一程序集中的任何代码都可以访问该类型或成员，但其他程序集中的代码不可以。

protected internal

由其声明的程序集或另一个程序集派生的类中任何代码都可访问的类型或成员。从另一个程序集进行访问必须在类声明中发生，该类声明派生自其中声明受保护的内部元素的类，并且必须通过派生的类类型的实例发生。

下面的示例演示如何为类型和成员指定访问修饰符：

```
public class Bicycle
{
    public void Pedal() { }
}
```

不是所有访问修饰符都可以在所有上下文中由所有类型或成员使用，在某些情况下类型成员的可访问性受到其包含类型的可访问性的限制。以下各节提供了有关可访问性的更多详细信息。

类和结构的可访问性

直接在命名空间中声明的类和结构（即，没有嵌套在其他类或结构中的类和结构）可以是公共类和结构，也可以是内部类和结构。如果不指定访问修饰符，则默认为 internal。

结构成员，包括嵌套的类和结构，可以声明为公共的、 内部的，或私人的。类成员（包括嵌套的类和结构）可以为公共的、受保护的内部、受保护的、内部的或私有的。类成员和结构成员的访问级别，包括嵌套类和结构，默认为私有。不可以从包含类型之外访问私有嵌套类型。

派生类的可访问性不能高于其基类型。换句话说，不能有从内部类 A 派生的公共类 B。如果允许这种情况，将会使 A 成为公共类，因为 A 的所有受保护的成员或内部成员都可以从派生类访问。

可以使用 InternalsVisibleToAttribute 使其他某些程序集能够访问您的内部类型。有关更多信息，请参见友元程序集（C# 和 Visual Basic）。

类成员和结构成员的可访问性

可以使用五种访问类型中的任何一种来声明类成员（包括嵌套的类和结构）。结构成员无法声明为受保护成员，因为结构不支持继承。

通常，成员的可访问性低于包含成员的类型的可访问性。如果由成员实现接口方法或重写已在公共基类中定义的虚拟方法时，内部类的公共成员可以从外部程序集访问。

任何成员的字段、 属性或事件的类型必须至少与该成员本身一样具备可访问性。同样，作为方法、索引器或代表的任一成员的返回类型和参数类型必须至少有与该成员本身一样的可访问性。例如，如果 C 不是公共类，则不能返回类 C 的公共方法 M。同样，如果 A 声明为私有，则类型 A 不能有受保护的属性。

用户定义的运算符必须始终声明为公共运算符。有关更多信息，请参见 运算符（C# 参考）。

析构函数不能具有可访问性修饰符。

要设置类成员或结构成员的访问级别，请向该成员声明添加适当的关键字，如下面的示例所示。

```
// public class:
public class Tricycle
{
    // protected method:
    protected void Pedal() { }

    // private field:
    private int wheels = 3;

    // protected internal property:
    protected internal int Wheels
    {
        get { return wheels; }
    }
}
```

> 注意
>
> 受保护内部可访问性级别的意思是受保护“或”内部，而不是受保护“和”内部。换句话说，可以从同一程序集内的任何类（包括派生类）中访问受保护的内部成员。若要限制为只有同一程序集内的派生类可以访问，请将类本身声明为内部，并将其成员声明为受保护。

其他类型

直接用命名空间声明时，可以将接口声明为公共接口或内部接口，只与类和结构一样，接口默认具有内部可访问性。接口成员始终是公共成员，因为接口的用途是让其他类型能够访问某个类或结构。访问修饰符不能应用于接口成员。

枚举成员始终是公共的，不能应用任何访问修饰符。

委托行为类似于类和结构。默认情况下，它们在命名空间中直接声明时具有内部访问权，在嵌套时具有私有访问权。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

类和结构（C# 编程指南）

接口（C# 编程指南）

private（C# 参考）

public（C# 参考）

internal（C# 参考）

protected（C# 参考）

class（C# 参考）

struct（C# 参考）

接口（C# 参考）

字段（C# 编程指南）

“字段”是直接在类或结构中声明的任何类型的变量。字段是其包含类型的“成员”。

类或结构可以拥有实例字段或静态字段，或同时拥有两者。实例字段特定于类型的实例。如果您拥有类 T 和实例字段 F，可以创建类型 T 的两个对象，并修改每个对象中 F 的值，这不影响另一对象中的该值。相比之下，静态字段属于类本身，在该类的所有实例中共享。从实例 A 所做的更改将立刻呈现在实例 B 和 C 上（如果它们访问该字段）。

通常应仅为具有私有或受保护可访问性的变量使用字段。您的类向客户端代码公开的数据应通过方法、属性和索引器提供。通过使用这些构造间接访问内部字段，可以针对无效的输入值提供防护。存储由公共属性公开的数据的私有字段称为“后备存储”或“支持字段”。

字段通常存储这样的数据：该数据必须可供多个类方法访问，并且其存储期必须长于任何单个方法的生存期。例如，表示日历日期的类可能有三个整数字段：一个表示月份，一个表示日期，还有一个表示年份。不在单个方法范围外部使用的变量应在方法体自身范围内声明为局部变量。

在类块中通过指定字段的访问级别，然后指定字段的类型，再指定字段的名称来声明这些字段。例如：

```
public class CalendarEntry
{
    // private field
    private DateTime date;

    // public field (Generally not recommended.)
    public string day;

    // Public property exposes date field safely.
    public DateTime Date
    {
        get
        {
            return date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth date
            if (value.Year > 1900 && value.Year <= DateTime.Today.Y
            {
                date = value;
            }
            else
                throw new ArgumentOutOfRangeException();
        }
```

```
    }

    // Public method also exposes date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            date = dt;
        }
        else
            throw new ArgumentOutOfRangeException();
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt != null && dt.Ticks < date.Ticks)
        {
            return date - dt;
        }
        else
            throw new ArgumentOutOfRangeException();

    }
}
```

若要访问对象中的字段，请在对象名称后面添加一个句点，然后添加该字段的名称，比如 objectname.fieldname。例如：

```
CalendarEntry birthday = new CalendarEntry();
birthday.day = "Saturday";
```

声明字段时可以使用赋值运算符为字段指定一个初始值。例如，若要自动将 "Monday" 赋给 day 字段，需要声明 day，如下例所示：

```
public class CalendarDateWithInitialization
{
    public string day = "Monday";
    //...
}
```

字段的初始化紧靠调用对象实例的构造函数之前。如果构造函数为字段赋值，则该值将覆盖字段声明期间给出的任何值。有关更多信息，请参见 使用构造函数（C# 编程指南）。

注意
字段初始值设定项不能引用其他实例字段。

字段可标记为 public、private、protected、internal 或 **protected internal**。这些访问修饰符定义类的使用者访问字段的方式。有关更多信息，请参见 访问修饰符（C# 编程指南）。

可以选择将字段声明为 static。这使得调用方在任何时候都能使用字段，即使类没有任何实例。有关更多信息，请参见 静态类和静态类成员（C# 编程指南）。

可以将字段声明为 readonly。只读字段只能在初始化期间或在构造函数中赋值。**static readonly** 字段非常类似于常数，只不过 C# 编译器不能在编译时访问静态只读字段的值，而只能在运行时访问。有关更多信息，请参见 常量（C# 编程指南）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

类和结构（C# 编程指南）

使用构造函数（C# 编程指南）

继承（C# 编程指南）

访问修饰符（C# 编程指南）

抽象类、密封类及类成员（C# 编程指南）

常量（C# 编程指南）

常量是在编译时已知并在程序的生存期内不发生更改的不可变值。常量使用 const 修饰符进行声明。只有 C# 内置类型（System.Object 除外）可以声明为 **const**。有关内置类型的列表，请参见内置类型表（C# 参考）。用户定义的类型（包括类、结构和数组）不能为 **const**。请使用 readonly 修饰符创建在运行时初始化一次即不可再更改的类、结构或数组。

C# 不支持 **const** 方法、属性或事件。

可以使用枚举类型为整数内置类型（例如 **int**、**uint**、**long** 等等）定义命名常量。有关更多信息，请参见 enum（C# 参考）。

常量必须在声明时初始化。例如：

```
class Calendar1
{
    public const int months = 12;
}
```

在此示例中，常量 months 始终为 12，不可更改，即使是该类自身也不能更改它。实际上，当编译器遇到 C# 源代码（例如 months）中的常量修饰符时，将直接把文本值替换到它生成的中间语言 (IL) 代码中。因为在运行时没有与常量关联的变量地址，所以 **const** 字段不能通过引用传递，并且不能在表达式中作为左值出现。

> 注意
>
> 当引用在其他代码如 DLL 中定义的常量值时应十分谨慎。如果新版本的 DLL 为常量定义了新的值，程序仍将保留旧的文本值，直到针对新版本重新编译程序。

可以同时声明多个相同类型的常量，例如：

```
class Calendar2
{
    const int months = 12, weeks = 52, days = 365;
}
```

如果不会造成循环引用，用于初始化一个常量的表达式可以引用另一个常量。例如：

```
class Calendar3
{
    const int months = 12;
    const int weeks = 52;
    const int days = 365;

    const double daysPerWeek = (double) days / (double) weeks;
    const double daysPerMonth = (double) days / (double) months;
}
```

常量可标记为 public、private、protected、internal 或 **protected****internal**。这些访问修饰符定义类的用户访问该常量的方式。有关更多信息，请参见 访问修饰符（C# 编程指南）。

因为常量值对该类型的所有实例是相同的，所以常量被当作 static 字段一样访问。不使用 **static** 关键字声明常量。未包含在定义常量的类中的表达式必须使用类名、一个句点和常量名来访问该常量。例如：

```
int birthstones = Calendar.months;
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

类和结构（C# 编程指南）

属性（C# 编程指南）

类型（C# 编程指南）

readonly（C# 参考）

Immutability in C# Part One: Kinds of Immutability（C# 中的不变性第一部分：不变性的种类）

属性（C# 编程指南）

属性是一种成员，它提供灵活的机制来读取、写入或计算私有字段的值。属性可用作公共数据成员，但它们实际上是称为“访问器”的特殊方法。这使得可以轻松访问数据，还有助于提高方法的安全性和灵活性。

在此示例中，TimePeriod 类存储时间段。该类在内部以秒为单位存储时间，但是名为 Hours 的属性允许客户端以小时为单位指定时间。 Hours 属性的访问器执行小时与秒之间的转换。

```
class TimePeriod
{
    private double seconds;

    public double Hours
    {
        get { return seconds / 3600; }
        set { seconds = value * 3600; }
    }
}

class Program
{
    static void Main()
    {
        TimePeriod t = new TimePeriod();

        // Assigning the Hours property causes the 'set' accessor t
        t.Hours = 24;

        // Evaluating the Hours property causes the 'get' accessor 
        System.Console.WriteLine("Time in hours: " + t.Hours);
    }
}
// Output: Time in hours: 24
```

表达式主体定义

直接只返回表达式结果的属性很常见。下面的语法快捷方式使用 **=>** 来定义这些属性：

```
public string Name => First + " " + Last;
```

属性必须为只读，并且你不能使用 **get** 访问器关键字。

属性概述

- 属性允许类公开获取和设置值的公共方法，而隐藏实现或验证代码。
- get 属性访问器用于返回属性值，而 set 访问器用于分配新值。这些访问器可以具有不同的访问级别。有关详细信息，请参阅限制访问器可访问性（C# 编程指南）。
- value 关键字用于定义由 **set** 访问器分配的值。
- 不实现 **set** 访问器的属性均为只读。
- 对于不需要任何自定义访问器代码的简单属性，请考虑选择使用自动实现的属性的选项。有关详细信息，请参阅自动实现的属性（C# 编程指南）。

相关章节

- 使用属性（C# 编程指南）
- 接口属性（C# 编程指南）
- 属性和索引器之间的比较（C# 编程指南）
- 限制访问器可访问性（C# 编程指南）
- 自动实现的属性（C# 编程指南）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

使用属性（C# 编程指南）

索引器（C# 编程指南）

方法（C# 编程指南）

方法是包含一系列语句的代码块。程序通过调用该方法并指定任何所需的方法参数使语句得以执行。在 C# 中，每个执行的指令均在方法的上下文中执行。Main 方法是每个 C# 应用程序的入口点，并在启动程序时由公共语言运行时 (CLR) 调用。

> 注意
>
> 本主题讨论命名的方法。有关匿名函数的信息，请参阅匿名函数（C# 编程指南）。

方法签名

通过指定访问级别（如 **public** 或 **private**）、可选修饰符（如 **abstract** 或 **sealed**）、返回值、方法的名称以及任何方法参数，在类或结构中声明方法。这些部件一起构成方法的签名。

> 注意
>
> 出于方法重载的目的，方法的返回类型不是方法签名的一部分。但是在确定委托和它所指向的方法之间的兼容性时，它是方法签名的一部分。

方法参数在括号内，并且用逗号分隔。空括号指示方法不需要任何参数。此类包含三种方法：

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method stat

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

方法访问

调用对象上的方法就像访问字段。在对象名之后添加一个句点、方法名和括号。参数列在括号里，并且用逗号分隔。因此，可在以下示例中调用 Motorcycle 类的方法：

```
class TestMotorcycle : Motorcycle
{

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

方法参数与参数

该方法定义指定任何所需参数的名称和类型。调用代码调用该方法时，它为每个参数提供了称为参数的具体值。参数必须与参数类型兼容，但调用代码中使用的参数名（如果有）不需要与方法中定义的参数名相同。例如:

```
public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaulates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}
```

按引用传递与按值传递

默认情况下，值类型传递给方法时，传递的是副本而不是对象本身。因此，对参数的更改不会影响调用方法中的原始副本。可以使用 ref 关键字按引用传递值类型。有关详细信息，请参阅传递值类型参数（C# 编程指南）。有关内置值类型的列表，请参阅值类型表（C# 参考）。

引用类型的对象传递到方法中时，将传递对对象的引用。也就是说，该方法接收的不是对象本身，而是指示该对象位置的参数。如果通过使用此引用更改对象的成员，即使是按值传递该对象，此更改也会反映在调用方法的参数中。

通过使用 **class** 关键字创建引用类型，如以下示例所示。

```
public class SampleRefType
{
    public int value;
}
```

现在，如果将基于此类型的对象传递到方法，则将传递对对象的引用。下面的示例将 SampleRefType 类型的对象传递到 ModifyObject 方法。

```
public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}
static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

该示例执行的内容实质上与先前示例相同，均按值将参数传递到方法。但是因为使用了引用类型，结果有所不同。 ModifyObject 中所做的对形参 obj 的 value 字段的修改，也会更改 TestRefType 方法中实参 rt 的 value 字段。 TestRefType 方法显示 33 作为输出。

有关如何按引用和按值传递引用类型的详细信息，请参阅传递引用类型参数（C# 编程指南）和引用类型（C# 参考）。

返回值

方法可以将值返回到调用方。如果列在方法名之前的返回类型不是 **void**，则该方法可通过使用 **return** 关键字返回值。带 **return** 关键字，后跟与返回类型匹配的值的语句将该值返回到方法调用方。 **return** 关键字还会停止执行该方法。如果返回类型为 **void**，没有值的 **return** 语句仍可用于停止执行该方法。没有 **return** 关键字，当方法到达代码块结尾时，将停止执行。具有非空的返回类型的方法都需要使用 **return** 关键字来返回值。例如，这两种方法都使用 **return** 关键字来返回整数：

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

若要使用从方法返回的值，调用方法可以在相同类型的值足够的地方使用该方法调用本身。也可以将返回值分配给变量。例如，以下两个代码示例实现了相同的目标：

```
  int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

在这种情况下，使用本地变量 result 存储值是可选的。此步骤可以帮助提高代码的可读性，或者如果需要存储该方法整个范围内参数的原始值，则此步骤可能很有必要。

有关详细信息，请参阅return（C# 参考）。

异步方法

通过使用异步功能，你可以调用异步方法而无需使用显式回调，也不需要跨多个方法或 lambda 表达式来手动拆分代码。Visual Studio 2012 中已引入异步功能。

如果用 async 修饰符标记方法，则可以使用该方法中的 await 运算符。当控件到达异步方法中的 await 表达式时，控件将返回到调用方，并在等待任务完成前，方法中进度将一直处于挂起状态。任务完成后，可以在方法中恢复执行。

> 注意
>
> 异步方法在遇到第一个尚未完成的 awaited 对象或到达异步方法的末尾时（以先发生者为准），将返回到调用方。

异步方法可以具有 Task<TResult>、Task 或 void 返回类型。Void 返回类型主要用于定义需要 void 返回类型的事件处理程序。无法等待返回 void 的异步方法，并且返回 void 方法的调用方无法捕获该方法引发的异常。

在以下示例中，DelayAsync 是具有 Task<TResult> 返回类型的异步方法。DelayAsync 具有返回整数的 **return** 语句。因此，DelayAsync 的方法声明必须具有 **Task<int>** 的返回类型。因为返回类型是 **Task<int>**，DoSomethingAsync 中 **await** 表达式的计算如以下语句所示得出整数：int result = await delayTask。

startButton_Click 方法是具有 void 返回类型的异步方法的示例。因为 DoSomethingAsync 是异步方法，调用 DoSomethingAsync 的任务必须等待，如以下语句所示：await DoSomethingAsync();。 startButton_Click 方法必须使用 **async** 修饰符进行定义，因为该方法具有 **await** 表达式。

```
// using System.Diagnostics;
// using System.Threading.Tasks;

// This Click event is marked with the async modifier.
private async void startButton_Click(object sender, RoutedEventArgs
{
    await DoSomethingAsync();
}

private async Task DoSomethingAsync()
{
    Task<int> delayTask = DelayAsync();
    int result = await delayTask;

    // The previous two statements may be combined into
    // the following statement.
    //int result = await DelayAsync();

    Debug.WriteLine("Result: " + result);
}

private async Task<int> DelayAsync()
{
    await Task.Delay(100);
    return 5;
}

// Output:
//  Result: 5
```

异步方法不能声明任何 ref 或 out 参数，但是可以调用具有这类参数的方法。

有关异步方法的详细信息，请参阅使用 Async 和 Await 的异步编程（C# 和 Visual Basic）、异步程序中的控制流（C# 和 Visual Basic）和异步返回类型（C# 和 Visual Basic）。

表达式主体定义

具有立即仅返回表达式结果，或单个语句作为方法主题的方法定义很常见。以下是使用 **=>** 定义此类方法的语法快捷方式：

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

如果该方法返回 **void** 或是异步方法，则该方法的主体必须是语句表达式（与 lambda 相同）。对于属性和索引器，两者必须是只读，并且不使用 **get** 访问器关键字。

迭代器

迭代器对集合执行自定义迭代，如列表或数组。迭代器使用 yield return 语句返回元素，每次返回一个。当 yield return 语句到达时，将记住当前在代码中的位置。下次调用迭代器时，将从该位置重新开始执行。

通过使用 foreach 语句从客户端代码调用迭代器。

迭代器的返回类型可以是 IEnumerable、IEnumerable<T>、IEnumerator 或 IEnumerator<T>。

有关详细信息，请参阅迭代器（C# 和 Visual Basic）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

类和结构（C# 编程指南）

访问修饰符（C# 编程指南）

静态类和静态类成员（C# 编程指南）

继承（C# 编程指南）

抽象类、密封类及类成员（C# 编程指南）

params（C# 参考）

return（C# 参考）

out（C# 参考）

ref（C# 参考）

传递参数（C# 编程指南）

构造函数（C# 编程指南）

任何时候，只要创建类或结构，就会调用它的构造函数。类或结构可能有多个接受不同参数的构造函数。构造函数使得程序员可设置默认值、限制实例化以及编写灵活且便于阅读的代码。有关更多信息和示例，请参见使用构造函数（C# 编程指南）和实例构造函数（C# 编程指南）。

如果您没有为对象提供构造函数，则默认情况下 C# 将创建一个构造函数，该构造函数实例化对象，并将成员变量设置为默认值表（C# 参考）中列出的默认值。有关更多信息和示例，请参见实例构造函数（C# 编程指南）。

静态类和结构也可以有构造函数。有关更多信息和示例，请参见静态构造函数（C# 编程指南）。

本节内容

使用构造函数（C# 编程指南）

实例构造函数（C# 编程指南）

私有构造函数（C# 编程指南）

静态构造函数（C# 编程指南）

如何：编写复制构造函数（C# 编程指南）

请参阅

C# 编程指南

类和结构（C# 编程指南）

析构函数（C# 编程指南）

static（C# 参考）

初始值设定项原因按相反顺序运行作为构造函数？Part One（第一部分）

析构函数（C# 编程指南）

析构函数用于析构类的实例。

备注

- 不能在结构中定义析构函数。只能对类使用析构函数。
- 一个类只能有一个析构函数。
- 无法继承或重载析构函数。
- 无法调用析构函数。它们是被自动调用的。
- 析构函数既没有修饰符，也没有参数。

例如，下面是类 Car 的析构函数的声明：

```
class Car
{
    ~Car()  // destructor
    {
        // cleanup statements...
    }
}
```

该析构函数隐式地对对象的基类调用 Finalize。这样，前面的析构函数代码被隐式地转换为以下代码：

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

这意味着对继承链中的所有实例递归地调用 Finalize 方法（从派生程度最大的到派生程度最小的）。

> 📝 注意
>
> 不应使用空析构函数。如果类包含析构函数，Finalize 队列中则会创建一个项。调用析构函数时，将调用垃圾回收器来处理该队列。如果析构函数为空，只会导致不必要的性能损失。

程序员无法控制何时调用析构函数，因为这是由垃圾回收器决定的。垃圾回收器检查是否存在应用程序不再使用的对象。如果垃圾回收器认为某个对象符合析构，则调用析构函数（如果有）并回收用来存储此对象的内存。程序退出时也会调用析构函数。

可以通过调用 Collect 强制进行垃圾回收，但大多数情况下应避免这样做，因为这样会导致性能问题。

使用析构函数释放资源

通常，与运行时不进行垃圾回收的开发语言相比，C# 无需太多的内存管理。这是因为 .NET Framework 垃圾回收器会隐式地管理对象的内存分配和释放。但是，当应用程序封装窗口、文件和网络连接这类非托管资源时，应当使用析构函数释放这些资源。当对象符合析构时，垃圾回收器将运行对象的 Finalize 方法。

资源的显式释放

如果您的应用程序在使用昂贵的外部资源，我们还建议您提供一种在垃圾回收器释放对象前显式地释放资源的方式。可通过实现来自 IDisposable 接口的 Dispose 方法来完成这一点，该方法为对象执行必要的清理。这样可大大提高应用程序的性能。即使有这种对资源的显式控制，析构函数也是一种保护措施，可用来在对 Dispose 方法的调用失败时清理资源。

有关清理资源的更多详细信息，请参见下列主题：

- Cleaning Up Unmanaged Resources
- Implementing a Dispose Method
- using 语句（C# 参考）

下面的示例创建三个类，这三个类构成了一个继承链。类 First 是基类，Second 是从 First 派生的，而 Third 是从 Second 派生的。这三个类都有析构函数。在 Main() 中，创建了派生程度最大的类的实例。注意：程序运行时，这三个类的析构函数将自动被调用，并且是按照从派生程度最大的到派生程度最小的次序调用。

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's destructor is c
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's destructor is
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's destructor is c
    }
}

class TestDestructors
{
    static void Main()
    {
        Third t = new Third();
    }

}
/* Output (to VS Output Window):
    Third's destructor is called.
    Second's destructor is called.
    First's destructor is called.
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

IDisposable

C# 编程指南

构造函数（C# 编程指南）

Garbage Collection

对象和集合初始值设定项（C# 编程指南）

使用对象初始值设定项，你可以在创建对象时向对象的任何可访问字段或属性分配值，而无需调用后跟赋值语句行的构造函数。利用对象初始值设定项语法，你可为构造函数指定参数或忽略参数（以及括号语法）。以下示例演示如何使用具有命名类型 Cat 的对象初始值设定项以及如何调用默认构造函数。请注意，自动实现的属性在 Cat 类中的用法。有关详细信息，请参阅自动实现的属性（C# 编程指南）。

```
  class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string Name { get; set; }
}
```

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
```

具有匿名类型的对象初始值设定项

尽管对象初始值设定项可用于任何上下文中，但它们在 LINQ 查询表达式中特别有用。查询表达式常使用只能通过使用对象初始值设定项进行初始化的匿名类型，如下面的声明所示。

```
var pet = new { Age = 10, Name = "Fluffy" };
```

利用匿名类型，**select** 查询表达式中的 LINQ 子句可以将原始序列的对象转换为其值和形状可能不同于原始序列的对象。如果你只想存储某个序列中每个对象的部分信息，则这很有用。在下面的示例中，假定产品对象 (p) 包含很多字段和方法，而你只想创建包含产品名和单价的对象序列。

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

执行此查询时，productInfos 变量将包含一系列对象，这些对象可以在 **foreach** 语句中进行访问，如下面的示例所示：

```
foreach(var p in productInfos){...}
```

新的匿名类型中的每个对象都具有两个公共属性，这两个属性接收与原始对象中的属性或字段相同的名称。你还可在创建匿名类型时重命名字段；下面的示例将 UnitPrice 字段重命名为 Price。

```
select new {p.ProductName, Price = p.UnitPrice};
```

具有可以为 null 的类型的对象初始值设定项

使用具有可以为 null 的结构的对象初始值设定项会导致编译时错误。

集合初始值设定项

集合初始值设定项允许在初始化实现 IEnumerable 的集合类或初始化具有 **Add** 扩展方法的类时，指定一个或多个元素初始值设定项。元素初始值设定项可以是简单的值、表达式或对象初始值设定项。通过使用集合初始值设定项，你将无需在源代码中指定对该类的 **Add** 方法的多个调用；编译器将添加这些调用。

下面的示例演示了两个简单的集合初始值设定项：

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

下面的集合初始值设定项使用对象初始值设定项来初始化上一个示例中定义的 Cat 类的对象。请注意，各个对象初始值设定项分别括在大括号中且用逗号隔开。

```
List<Cat> cats = new List<Cat>
{
    new Cat(){ Name = "Sylvester", Age=8 },
    new Cat(){ Name = "Whiskers", Age=2 },
    new Cat(){ Name = "Sasha", Age=14 }
};
```

如果集合的 null 方法允许，则可以将 **Add** 指定为集合初始值设定项中的一个元素。

```
List<Cat> moreCats = new List<Cat>
{
    new Cat(){ Name = "Furrytail", Age=5 },
    new Cat(){ Name = "Peaches", Age=4 },
    null
};
```

如果集合支持索引，可以指定索引元素。

```
var numbers = new Dictionary<int, string> {
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

```
// The following code consolidates examples from the topic.
class ObjInitializers
{
    class Cat
    {
        // Auto-implemented properties.
        public int Age { get; set; }
        public string Name { get; set; }
    }

    static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };

        List<Cat> cats = new List<Cat>
        {
            new Cat(){ Name = "Sylvester", Age=8 },
            new Cat(){ Name = "Whiskers", Age=2 },
            new Cat(){ Name = "Sasha", Age=14 }
        };

        List<Cat> moreCats = new List<Cat>
        {
            new Cat(){ Name = "Furrytail", Age=5 },
            new Cat(){ Name = "Peaches", Age=4 },
            null
        };

        // Display results.
        System.Console.WriteLine(cat.Name);

        foreach (Cat c in cats)
            System.Console.WriteLine(c.Name);

        foreach (Cat c in moreCats)
            if (c != null)
                System.Console.WriteLine(c.Name);
            else
                System.Console.WriteLine("List element has null va
    }
    // Output:
    //Fluffy
    //Sylvester
    //Whiskers
    //Sasha
    //Furrytail
    //Peaches
    //List element has null value.
}
```

请参阅

C# 编程指南

LINQ 查询表达式（C# 编程指南）

匿名类型（C# 编程指南）

如何：使用 foreach 访问集合类（C# 编程指南）

下面的代码示例演示如何编写可与 foreach 结合使用的非泛型集合类。该示例定义了字符串 tokenizer 类。

> 注意
>
> 此示例描述的是仅当您无法使用泛型集合类时才采用的推荐做法。有关如何实现支持 IEnumerable<T> 的类型安全的泛型集合类，请参见迭代器（C# 和 Visual Basic）。

在该示例中，以下代码段使用 Tokens 类通过" "和"-"分隔符将句子"This is a sample sentence."分成若干标记。该代码然后使用 **foreach** 语句显示这些标记。

```
Tokens f = new Tokens("This is a sample sentence.", new char[] {'

// Display the tokens.
foreach (string item in f)
{
    System.Console.WriteLine(item);
}
```

在内部，Tokens 类使用数组存储这些标记。因为数组可实现 IEnumerator 和 IEnumerable，所以代码示例使用了数组的枚举方法（GetEnumerator、MoveNext、Reset 和 Current），而不是在 Tokens 类中定义这些方法。方法定义包括在该示例中，以明确如何定义它们以及每个定义的内容。

```
using System.Collections;

// Declare the Tokens class. The class implements the IEnumerable i
public class Tokens : IEnumerable
{
    private string[] elements;

    Tokens(string source, char[] delimiters)
    {
        // The constructor parses the string argument into tokens.
        elements = source.Split(delimiters);
    }

    // The IEnumerable interface requires implementation of method
    public IEnumerator GetEnumerator()
    {
        return new TokenEnumerator(this);
    }
```

```
    // Declare an inner class that implements the IEnumerator inter
    private class TokenEnumerator : IEnumerator
    {
        private int position = -1;
        private Tokens t;

        public TokenEnumerator(Tokens t)
        {
            this.t = t;
        }

        // The IEnumerator interface requires a MoveNext method.
        public bool MoveNext()
        {
            if (position < t.elements.Length - 1)
            {
                position++;
                return true;
            }
            else
            {
                return false;
            }
        }

        // The IEnumerator interface requires a Reset method.
        public void Reset()
        {
            position = -1;
        }

        // The IEnumerator interface requires a Current method.
        public object Current
        {
            get
            {
                return t.elements[position];
            }
        }
    }

    // Test the Tokens class.
    static void Main()
    {
        // Create a Tokens instance.
        Tokens f = new Tokens("This is a sample sentence.", new cha

        // Display the tokens.
        foreach (string item in f)
        {
            System.Console.WriteLine(item);
        }
    }
```

```
}
/* Output:
    This
    is
    a
    sample
    sentence.
*/
```

在 C# 中，集合类不必通过实现 IEnumerable 和 IEnumerator 来与 **foreach** 兼容。如果此类具有所需的 GetEnumerator、MoveNext、Reset 和 Current 成员，则可与 **foreach** 结合使用。省略接口有一个好处：即，您可以比 Object 更为具体地定义 Current 的返回类型。这会提供类型安全。

例如，可更改上述示例中的以下行。

```
// Change the Tokens class so that it no longer implements IEnumera
public class Tokens
{
    // . . .

    // Change the return type for the GetEnumerator method.
    public TokenEnumerator GetEnumerator()
    {   }

    // Change TokenEnumerator so that it no longer implements IEnur
    public class TokenEnumerator
    {
        // . . .

        // Change the return type of method Current to string.
        public string Current
        {   }
    }
 }
```

由于 Current 返回字符串，因此编译器能够检测何时在 **foreach** 语句中使用了不兼容的类型，如以下代码所示。

```
// Error: Cannot convert type string to int.
foreach (int item in f)
```

省略 IEnumerable 和 IEnumerator 的缺点是：集合类不再与其他公共语言运行时语言的 **foreach** 语句或等效语句交互。

请参阅

System.Collections.Generic

C# 参考

C# 编程指南

数组（C# 编程指南）

集合（C# 和 Visual Basic）

嵌套类型（C# 编程指南）

在类或结构内部定义的类型称为嵌套类型。例如：

```
class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

不管外部类型是类还是结构，嵌套类型均默认为 private，但是可以设置为 public、protected internal、protected、internal 或 private。在上面的示例中，Nested 对外部类型是不可访问的，但可以设置为 public，如下所示：

```
class Container
{
    public class Nested
    {
        Nested() { }
    }
}
```

嵌套类型（或内部类型）可访问包含类型（或外部类型）。若要访问包含类型，请将其作为构造函数传递给嵌套类型。例如：

```
public class Container
{
    public class Nested
    {
        private Container parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

嵌套类型可以访问其包含类型可以访问的所有成员。它可以访问包含类型的私有成员和受保护成员（包括所有继承的受保护成员）。

在前面的声明中，类 Nested 的完整名称为 Container.Nested。这是用来创建嵌套类新实例的名称，如下所示：

```
Container.Nested nest = new Container.Nested();
```

请参阅

C# 编程指南

类和结构（C# 编程指南）

访问修饰符（C# 编程指南）

构造函数（C# 编程指南）

分部类和方法（C# 编程指南）

可以将类或结构、接口或方法的定义拆分到两个或多个源文件中。每个源文件包含类型或方法定义的一部分，编译应用程序时将把所有部分组合起来。

分部类

在以下几种情况下需要拆分类定义：

- 处理大型项目时，使一个类分布于多个独立文件中可以让多位程序员同时对该类进行处理。
- 使用自动生成的源时，无需重新创建源文件便可将代码添加到类中。Visual Studio 在创建 Windows 窗体、Web 服务包装代码等时都使用此方法。无需修改 Visual Studio 创建的文件，就可创建使用这些类的代码。
- 若要拆分类定义，请使用 partial 关键字修饰符，如下所示：

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

partial 关键字指示可在命名空间中定义该类、结构或接口的其他部分。所有部分都必须使用 **partial** 关键字。在编译时，各个部分都必须可用来形成最终的类型。各个部分必须具有相同的可访问性，如 **public**、**private** 等。

如果将任意部分声明为抽象的，则整个类型都被视为抽象的。如果将任意部分声明为密封的，则整个类型都被视为密封的。如果任意部分声明基类型，则整个类型都将继承该类。

指定基类的所有部分必须一致，但忽略基类的部分仍继承该基类型。各个部分可以指定不同的基接口，最终类型将实现所有分部声明所列出的全部接口。在某一分部定义中声明的任何类、结构或接口成员可供所有其他部分使用。最终类型是所有部分在编译时的组合。

📝 注意
partial 修饰符不可用于委托或枚举声明中。

下面的示例演示嵌套类型可以是分部的，即使它们所嵌套于的类型本身并不是分部的也如此。

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }
    partial class Nested
    {
        void Test2() { }
    }
}
```

编译时将对分部类型定义的特性进行合并。例如，请考虑下列声明：

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

它们等效于以下声明：

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

将从所有分部类型定义中对以下内容进行合并：

- XML 注释
- 接口
- 泛型类型参数特性
- 类特性
- members

例如，请考虑下列声明：

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

它们等效于以下声明：

```
class Earth : Planet, IRotate, IRevolve { }
```

限制

处理分部类定义时需遵循下面的几个规则：

- 要作为同一类型的各个部分的所有分部类型定义都必须使用 **partial** 进行修饰。例如，下面的类声明将生成错误：

  ```
  public partial class A { }
  //public class A { }  // Error, must also be marked partial
  ```

- **partial** 修饰符只能出现在紧靠关键字 **class**、**struct** 或 **interface** 前面的位置。
- 分部类型定义中允许使用嵌套的分部类型，如下面的示例中所示：

  ```
  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }

  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }
  ```

- 要成为同一类型的各个部分的所有分部类型定义都必须在同一程序集和同一模块（.exe 或 .dll 文件）中进行定义。分部定义不能跨越多个模块。
- 类名和泛型类型参数在所有的分部类型定义中都必须匹配。泛型类型可以是分部的。每个分部声明都必须以相同的顺序使用相同的参数名。
- 下面的用于分部类型定义中的关键字是可选的，但是如果某关键字出现在一个分部类型定义中，则该关键字不能与在同一类型的其他分部定义中指定的关键字冲突：
  - public
  - private

- protected
- internal
- abstract
- sealed
- 基类
- new 修饰符（嵌套部分）
- 泛型约束

  有关更多信息，请参见 类型参数的约束（C# 编程指南）。

示例 1

说明

下面的示例在一个分部类定义中声明 CoOrds 类的字段和构造函数，在另一个分部类定义中声明成员 PrintCoOrds。

代码

```
public partial class CoOrds
{
    private int x;
    private int y;

    public CoOrds(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class CoOrds
{
    public void PrintCoOrds()
    {
        Console.WriteLine("CoOrds: {0},{1}", x, y);
    }

}

class TestCoOrds
{
    static void Main()
    {
        CoOrds myCoOrds = new CoOrds(10, 15);
        myCoOrds.PrintCoOrds();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: CoOrds: 10,15
```

示例 2

说明

从下面的示例可以看出，您也可以开发分部结构和接口。

代码

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

分部方法

分部类或结构可以包含分部方法。类的一个部分包含方法的签名。可以在同一部分或另一个部分中定义可选实现。如果未提供该实现，则会在编译时移除方法以及对方法的所有调用。

分部方法使类的某个部分的实施者能够定义方法（类似于事件）。类的另一部分的实施者可以决定是否实现该方法。如果未实现该方法，编译器将移除方法签名以及对该方法的所有调用。调用该方法，包括调用中的任何计算结果，在运行时没有任何影响。因此，分部类中的任何代码都可以随意地使用分部方法，即使未提供实现也是如此。如果调用了未实现的方法，将不会导致编译时错误或运行时错误。

在自定义生成的代码时，分部方法特别有用。这些方法允许保留方法名称和签名，因此生成的代码可以调用方法，而开发人员可以决定是否实现方法。与分部类非常类似，分部方法使代码生成器创建的代码和开发人员创建的代码能够协同工作，而不会产生运行时开销。

分部方法声明由两个部分组成：定义和实现。它们可以位于分部类的不同部分中，也可以位于同一部分中。如果不存在实现声明，则编译器将优化定义声明和对方法的所有调用。

```
// Definition in file1.cs
partial void onNameChanged();

// Implementation in file2.cs
partial void onNameChanged()
{
  // method body
}
```

- 分部方法声明必须以上下文关键字 partial 开头，并且方法必须返回 void。
- 分部方法可以有 ref 参数，但不能有 out 参数。
- 分部方法为隐式 private 方法，因此不能为 virtual 方法。
- 分部方法不能为 extern 方法，因为主体的存在确定了方法是在定义还是在实现。
- 分部方法可以有 static 和 unsafe 修饰符。
- 分部方法可以为泛型的。约束将放在定义分部方法声明上，但也可以选择重复放在实现声明上。参数和类型参数名称在实现声明和定义声明中不必相同。
- 您可以为已定义并实现的分部方法生成委托，但不能为已经定义但未实现的分部方法生成委托。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

类（C# 编程指南）

结构（C# 编程指南）

接口（C# 编程指南）

分部（类型）（C# 参考）

匿名类型（C# 编程指南）

匿名类型提供了一种方便的方法，可用来将一组只读属性封装到单个对象中，而无需首先显式定义一个类型。类型名由编译器生成，并且不能在源代码级使用。每个属性的类型由编译器推断。

可通过使用 new 运算符和对象初始值创建匿名类型。有关对象初始值设定项的详细信息，请参阅对象和集合初始值设定项（C# 编程指南）。

以下示例显示了用两个名为 Amount 和 Message 的属性进行初始化的匿名类型。

```
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the follov
// statement to verify that their inferred types are int and strin
Console.WriteLine(v.Amount + v.Message);
```

匿名类型通常用在查询表达式的 select 子句中，以便返回源序列中每个对象的属性子集。有关查询的详细信息，请参阅LINQ 查询表达式（C# 编程指南）。

匿名类型包含一个或多个公共只读属性。包含其他种类的类成员（如方法或事件）为无效。用来初始化属性的表达式不能为 **null**、匿名函数或指针类型。

最常见的方案是用其他类型的属性初始化匿名类型。在下面的示例中，假定名为 Product 的类存在。类 Product 包括 Color 和 Price 属性，以及你不感兴趣的其他属性。变量 products 是 Product 对象的集合。匿名类型声明以 **new** 关键字开始。声明初始化了一个只使用 Product 的两个属性的新类型。这将导致在查询中返回较少数量的数据。

如果你没有在匿名类型中指定成员名称，编译器会为匿名类型成员指定与用于初始化这些成员的属性相同的名称。必须为使用表达式初始化的属性提供名称，如下面的示例所示。在下面示例中，匿名类型的属性名称都为 Color 和 Price。

```
var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```

通常，当你使用匿名类型来初始化变量时，可以通过使用 var 将变量作为隐式键入的本地变来变量进行声明。类型名称无法在变量声明中给出，因为只有编译器能访问匿名类型的基础名称。有关 **var**的详细信息，请参阅隐式类型的局部变量（C# 编程指南）。

可通过将隐式键入的本地变量与隐式键入的数组相结合创建匿名键入的元素的数组，如下面的示例所示。

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { nar
```

备注

匿名类型是直接从对象派生的类类型，并且其无法强制转换为除对象外的任意类型。虽然你的应用程序不能访问它，编译器还是提供了每一个匿名类型的名称。从公共语言运行时的角度来看，匿名类型与任何其他引用类型没有什么不同。

如果程序集中的两个或多个匿名对象初始值指定了属性序列，这些属性采用相同顺序且具有相同的名称和类型，则编译器将对象视为相同类型的实例。它们共享同一编译器生成的类型信息。

无法将字段、属性、时间或方法的返回类型声明为具有匿名类型。同样，你不能将方法、属性、构造函数或索引器的形参声明为具有匿名类型。要将匿名类型或包含匿名类型的集合作为参数传递给某一方法，可将参数作为类型对象进行声明。但是，这样做会使强类型化作用无效。如果必须存储查询结果或者必须将查询结果传递到方法边界外部，请考虑使用普通的命名结构或类而不是匿名类型。

由于匿名类型上的 Equals 和 GetHashCode 方法是根据方法属性的 **Equals** 和 **GetHashCode** 定义的，因此仅当同一匿名类型的两个实例的所有属性都相等时，这两个实例才相等。

请参阅

C# 编程指南

对象和集合初始值设定项（C# 编程指南）

Getting Started with LINQ in C#

LINQ 查询表达式（C# 编程指南）

委托（C# 编程指南）

delegate 是表示对具有特定参数列表和返回类型的方法的引用的类型。在实例化委托时，你可以将其实例与任何具有兼容签名和返回类型的方法相关联。你可以通过委托实例调用方法。

委托用于将方法作为参数传递给其他方法。事件处理程序就是通过委托调用的方法。你可以创建一个自定义方法，当发生特定事件时，某个类（如 Windows 控件）就可以调用你的方法。下面的示例演示了一个委托声明：

```
public delegate int PerformCalculation(int x, int y);
```

可将任何可访问类或结构中与委托类型匹配的任何方法分配给委托。该方法可以是静态方法，也可以是实例方法。这样便能通过编程方式来更改方法调用，还可以向现有类中插入新代码。

注意

在方法重载的上下文中，方法的签名不包括返回值。但在委托的上下文中，签名包括返回值。换句话说，方法和委托必须具有相同的返回类型。

将方法作为参数进行引用的能力使委托成为定义回调方法的理想选择。例如，对比较两个对象的方法的引用可以作为参数传递到排序算法中。由于比较代码在一个单独的过程中，因此可通过更常见的方式编写排序算法。

委托概述

委托具有以下属性：

- 委托类似于 C++ 函数指针，但它们是类型安全的。
- 委托允许将方法作为参数进行传递。
- 委托可用于定义回调方法。
- 委托可以链接在一起；例如，可以对一个事件调用多个方法。
- 方法不必与委托类型完全匹配。有关详细信息，请参阅在委托中使用变体（C# 和 Visual Basic）。
- C# 2.0 版引入了匿名方法的概念，此类方法允许将代码块作为参数传递来代替单独定义的方法。C# 3.0 引入了 Lambda 表达式，利用它们可以更简练地编写内联代码块。匿名方法和 Lambda 表达式（在某些上下文中）都可编译为委托类型。这些功能现在统称为匿名函数。有关 lambda 表达式的更多信息，请参见 匿名函数（C# 编程指南）。

本节内容

- 使用委托（C# 编程指南）
- When to Use Delegates Instead of Interfaces (C# Programming Guide)
- 带有命名方法的委托与带有匿名方法的委托（C# 编程指南）
- 匿名方法（C# 编程指南）
- 在委托中使用变体（C# 和 Visual Basic）
- 如何：合并委托（多路广播委托）（C# 编程指南）
- 如何：声明、实例化和使用委托（C# 编程指南）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

重要章节

http://go.microsoft.com/fwlink/?LinkId=195395

http://go.microsoft.com/fwlink/?LinkId=195369

http://go.microsoft.com/fwlink/?LinkId=195418

http://go.microsoft.com/fwlink/?LinkId=195412

请参阅

Delegate

C# 编程指南

事件（C# 编程指南）

使用委托（C# 编程指南）

委托是安全封装方法的类型，类似于 C 和 C++ 中的函数指针。与 C 函数指针不同的是，委托是面向对象的、类型安全的和可靠的。委托的类型由委托的名称确定。以下示例声明名为 Del 的委托，该委托可以封装采用字符串作为参数并返回 void 的方法：

```
public delegate void Del(string message);
```

委托对象通常通过提供委托将封装的方法的名称或使用匿名方法构造。对委托进行实例化后，委托会将对其进行的方法调用传递到该方法。调用方传递到委托的参数将传递到该方法，并且委托会将方法的返回值（如果有）返回到调用方。这被称为调用委托。实例化的委托可以按封装的方法本身进行调用。例如：

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    System.Console.WriteLine(message);
}
```

```
// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

委托类型派生自 .NET Framework 中的 Delegate 类。委托类型是封装的，它们不能派生出其他类，也不能从 Delegate 派生出自定义类。由于实例化的委托是一个对象，因此可以作为参数传递或分配给一个属性。这允许方法作为参数接受委托并在稍后调用委托。这被称为异步回调，是在长进程完成时通知调用方的常用方法。当以这种方式使用委托时，使用委托的代码不需要知道要使用的实现方法。功能类似于封装接口提供的功能。

回调的另一个常见用途是定义自定义比较方法并将该委托传递到短方法。它允许调用方的代码成为排序算法的一部分。以下示例方法使用 Del 类型作为参数：

```
public void MethodWithCallback(int param1, int param2, Del callbac
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

然后，你可以将上面创建的委托传递到该方法：

```
MethodWithCallback(1, 2, handler);
```

并将以下输出接收到控制台：

The number is: 3

以抽象方式使用委托时，MethodWithCallback 不需要直接调用控制台，记住，其不必设计为具有控制台。 MethodWithCallback 的作用是简单准备字符串并将字符串传递到其他方法。由于委托的方法可以使用任意数量的参数，此功能特别强大。

当委托构造为封装实例方法时，委托将同时引用实例和方法。委托不知道除其所封装方法以外的实例类型，因此委托可以引用任何类型的对象，只要该对象上有与委托签名匹配的方法。当委托构造为封装静态方法时，委托仅引用方法。请考虑以下声明：

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

加上之前显示的静态 DelegateMethod，我们现在已有三个 Del 实例可以封装的方法。

调用时，委托可以调用多个方法。这被称为多播。若要向委托的方法列表（调用列表）添加其他方法，只需使用加法运算符或加法赋值运算符（“+”或“+=”）添加两个委托。例如：

```
MethodClass obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

此时，allMethodsDelegate 的调用列表中包含三个方法，分别为 Method1、Method2 和 DelegateMethod。原有的三个委托（d1、d2 和 d3）保持不变。调用 allMethodsDelegate 时，将按顺序调用所有三个方法。如果委托使用引用参数，引用将按相反的顺序传递到所有这三个方法，并且一种方法进行的任何更改都将在另一种方法上见到。当方法引发未在方法内捕获到的异常时，该异常将传递到委托的调用方，并且不会调用调用列表中的后续方法。如果委托具有返回值和/或输出参数，它将返回上次调用方法的返回值和参数。若要删除调用列表中的方法，请使用减法运算符或减法赋值运算符（“+”或“+=”）。例如：

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Del oneMethodDelegate = allMethodsDelegate - d2;
```

由于委托类型派生自 **System.Delegate**，因此可以在委托上调用该类定义的方法和属性。例如，若要查询委托调用列表中方法的数量，你可以编写：

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

调用列表中具有多个方法的委托派生自 MulticastDelegate，该类属于 **System.Delegate** 的子类。由于这两个类都支持 **GetInvocationList**，因此在其他情况下，上述代码也将产生作用。

多播委托广泛用于事件处理中。事件源对象将事件通知发送到已注册接收该事件的接收方对象。若要注册一个事件，接收方需要创建用于处理该事件的方法，然后为该方法创建委托并将委托传递到事件源。事件发生时，源调用委托。然后，委托将对接收方调用事件处理方法，从而提供事件数据。给定事件的委托类型由事件源确定。有关详细信息，请参阅事件（C# 编程指南）。

在编译时比较分配的两个不同类型的委托将导致编译错误。如果委托实例是静态的 **System.Delegate** 类型，则允许比较，但在运行时将返回 false。例如：

```
delegate void Delegate1();
delegate void Delegate2();

static void method(Delegate1 d, Delegate2 e, System.Delegate f)
{
    // Compile-time error.
    //Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    System.Console.WriteLine(d == f);
}
```

请参阅

C# 编程指南

委托（C# 编程指南）

在委托中使用变体（C# 和 Visual Basic）

委托中的变体（C# 和 Visual Basic）

对 Func 和 Action 泛型委托使用变体（C# 和 Visual Basic）

事件（C# 编程指南）

带有命名方法的委托与带有匿名方法的委托（C# 编程指南）

委托可以与命名方法关联。使用命名方法对委托进行实例化时，该方法将作为参数传递，例如：

```
// Declare a delegate:
delegate void Del(int x);

// Define a named method:
void DoWork(int k) { /* ... */ }

// Instantiate the delegate using the method as a parameter:
Del d = obj.DoWork;
```

这被称为使用命名的方法。使用命名方法构造的委托可以封装静态方法或实例方法。在早期版本的 C# 中，命名方法是对委托进行实例化的唯一方式。但是，在不希望付出创建新方法的系统开销时，C# 使您可以对委托进行实例化，并立即指定委托在被调用时将处理的代码块。代码块可以包含 lambda 表达式或匿名方法。有关更多信息，请参见 匿名函数（C# 编程指南）。

备注

作为委托参数传递的方法必须与委托声明具有相同的签名。

委托实例可以封装静态或实例方法。

尽管委托可以使用 out 参数，但建议您不要将其用于多路广播事件委托，因为您无法知道哪个委托将被调用。

示例 1

以下是声明及使用委托的一个简单示例。注意，委托 Del 和关联的方法 MultiplyNumbers 具有相同的签名

```
// Declare a delegate
delegate void Del(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        Del d = m.MultiplyNumbers;

        // Invoke the delegate object.
        System.Console.WriteLine("Invoking the delegate using 'Mult
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        System.Console.Write(m * n + " ");
    }
}
/* Output:
    Invoking the delegate using 'MultiplyNumbers':
    2 4 6 8 10
*/
```

示例 2

在下面的示例中，一个委托被同时映射到静态方法和实例方法，并分别返回特定的信息。

```
// Declare a delegate
delegate void Del();

class SampleClass
{
    public void InstanceMethod()
    {
        System.Console.WriteLine("A message from the instance metho
    }

    static public void StaticMethod()
    {
        System.Console.WriteLine("A message from the static method
    }
}

class TestSampleClass
{
    static void Main()
    {
        SampleClass sc = new SampleClass();

        // Map the delegate to the instance method:
        Del d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
    A message from the instance method.
    A message from the static method.
*/
```

请参阅

C# 编程指南

委托（C# 编程指南）

匿名方法（C# 编程指南）

如何：合并委托（多路广播委托）（C# 编程指南）

事件（C# 编程指南）

如何：合并委托（多路广播委托）（C# 编程指南）

本示例演示如何创建多播委托。 委托对象的一个有用属性是：可以使用 **+** 运算符将多个对象分配给一个委托实例。多播委托包含已分配委托的列表。在调用多播委托时，它会按顺序调用列表中的委托。只能合并相同类型的委托。

- 运算符可用于从多播委托中移除组件委托。

```
using System;

// Define a custom delegate that has a string parameter and returns
delegate void CustomDel(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomDel
    static void Hello(string s)
    {
        System.Console.WriteLine("  Hello, {0}!", s);
    }

    static void Goodbye(string s)
    {
        System.Console.WriteLine("  Goodbye, {0}!", s);
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Create the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Create the delegate object byeDel that references the
        // method Goodbye.
        byeDel = Goodbye;

        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel
        // which calls only the method Goodbye.
```

```
        multiMinusHiDel = multiDel - hiDel;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}
/* Output:
Invoking delegate hiDel:
  Hello, A!
Invoking delegate byeDel:
  Goodbye, B!
Invoking delegate multiDel:
  Hello, C!
  Goodbye, C!
Invoking delegate multiMinusHiDel:
  Goodbye, D!
*/
```

请参阅

MulticastDelegate

C# 编程指南

事件（C# 编程指南）

如何：声明、实例化和使用委托（C# 编程指南）

在 C# 1.0 及更高版本中，可以按以下示例所示声明委托。

```
// Declare a delegate.
delegate void Del(string str);

// Declare a method with the same signature as the delegate.
static void Notify(string name)
{
    Console.WriteLine("Notification received for: {0}", name);
}
```

```
// Create an instance of the delegate.
Del del1 = new Del(Notify);
```

C# 2.0 提供了更简单的方法来编写上面的声明，如以下示例所示。

```
// C# 2.0 provides a simpler way to declare an instance of Del.
Del del2 = Notify;
```

在 C# 2.0 及更高版本中，还可以使用匿名方法来声明和初始化委托，如以下示例所示。

```
// Instantiate Del by using an anonymous method.
Del del3 = delegate(string name)
    { Console.WriteLine("Notification received for: {0}", name); }
```

在 C# 3.0 及更高版本中，还可以使用 Lambda 表达式来声明和实例化委托，如以下示例所示。

```
// Instantiate Del by using a lambda expression.
Del del4 = name =>  { Console.WriteLine("Notification received for
```

有关更多信息，请参见 Lambda 表达式（C# 编程指南）。

下面的示例阐释声明、实例化和使用委托。 BookDB 类封装一个书店数据库，它维护一个书籍数据库。它公开 ProcessPaperbackBooks 方法，该方法在数据库中查找所有平装书，并对每本平装书调用一个委托。使用的 **delegate** 类型名为 ProcessBookDelegate。 Test 类使用该类打印平装书的书名和平均价格。

委托的使用促进了书店数据库和客户代码之间功能的良好分隔。客户代码不知道书籍的存储方式和书店代码查找平装书的方式。书店代码也不知道找到平装书后将对平装书执行什么处理。

```
// A set of classes for handling a bookstore:
namespace Bookstore
{
    using System.Collections;

    // Describes a book in the book list:
    public struct Book
    {
        public string Title;        // Title of the book.
        public string Author;       // Author of the book.
        public decimal Price;       // Price of the book.
        public bool Paperback;      // Is it paperback?

        public Book(string title, string author, decimal price, boo
        {
            Title = title;
            Author = author;
            Price = price;
            Paperback = paperBack;
        }
    }

    // Declare a delegate type for processing a book:
    public delegate void ProcessBookDelegate(Book book);

    // Maintains a book database.
    public class BookDB
    {
        // List of all books in the database:
        ArrayList list = new ArrayList();

        // Add a book to the database:
        public void AddBook(string title, string author, decimal pr
        {
            list.Add(new Book(title, author, price, paperBack));
        }

        // Call a passed-in delegate on each paperback book to proc
        public void ProcessPaperbackBooks(ProcessBookDelegate proce
        {
            foreach (Book b in list)
            {
                if (b.Paperback)
```

```
                    // Calling the delegate:
                    processBook(b);
                }
            }
        }
    }

// Using the Bookstore classes:
namespace BookTestClient
{
    using Bookstore;

    // Class to total and average prices of books:
    class PriceTotaller
    {
        int countBooks = 0;
        decimal priceBooks = 0.0m;

        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
        }

        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }

    // Class to test the book database:
    class TestBookDB
    {
        // Print the title of the book.
        static void PrintTitle(Book b)
        {
            System.Console.WriteLine("   {0}", b.Title);
        }

        // Execution starts here.
        static void Main()
        {
            BookDB bookDB = new BookDB();

            // Initialize the database with some books:
            AddBooks(bookDB);

            // Print all the titles of paperbacks:
            System.Console.WriteLine("Paperback Book Titles:");

            // Create a new delegate object associated with the sta
            // method Test.PrintTitle:
            bookDB.ProcessPaperbackBooks(PrintTitle);
```

```
            // Get the average price of a paperback by using
            // a PriceTotaller object:
            PriceTotaller totaller = new PriceTotaller();

            // Create a new delegate object associated with the nor
            // method AddBookToTotal on the object totaller:
            bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);

            System.Console.WriteLine("Average Paperback Book Price
                    totaller.AveragePrice());
        }

        // Initialize the book database with some test books:
        static void AddBooks(BookDB bookDB)
        {
            bookDB.AddBook("The C Programming Language", "Brian W.
            bookDB.AddBook("The Unicode Standard 2.0", "The Unicode
            bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan",
            bookDB.AddBook("Dogbert's Clues for the Clueless", "Sco
        }
    }
}
/* Output:
Paperback Book Titles:
   The C Programming Language
   The Unicode Standard 2.0
   Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/
```

可靠编程

- 声明委托。

  下面的语句声明一个新的委托类型。

  ```
  public delegate void ProcessBookDelegate(Book book);
  ```

  每个委托类型都描述参数的数目和类型，以及它可以封装的方法的返回值类型。每当需要一组新的参数类型或新的返回值类型时，都必须声明一个新的委托类型。

- 实例化委托。

  声明了委托类型后，必须创建委托对象并使之与特定方法关联。在上一个示例中，您通过按下面示例中的方式将 PrintTitle 方法传递到 ProcessPaperbackBooks 方法来实现这一点：

```
bookDB.ProcessPaperbackBooks(PrintTitle);
```

这将创建与静态方法 Test.PrintTitle 关联的新委托对象。类似地，对象 totaller 的非静态方法 AddBookToTotal 是按下面示例中的方式传递的：

```
bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
```

在两个示例中，都向 ProcessPaperbackBooks 方法传递了一个新的委托对象。

委托创建后，它的关联方法就不能更改；委托对象是不可变的。

- 调用委托。

  创建委托对象后，通常将委托对象传递给将调用该委托的其他代码。通过委托对象的名称（后面跟着要传递给委托的参数，括在括号内）调用委托对象。下面是委托调用的示例：

  ```
  processBook(b);
  ```

  与本例一样，可以通过使用 **BeginInvoke** 和 **EndInvoke** 方法同步或异步调用委托。

请参阅

C# 编程指南

事件（C# 编程指南）

委托（C# 编程指南）

枚举类型（C# 编程指南）

枚举类型（也称为枚举）为定义一组可以赋给变量的命名整数常量提供了一种有效的方法。例如，假设您必须定义一个变量，该变量的值表示一周中的一天。该变量只能存储七个有意义的值。若要定义这些值，可以使用枚举类型。枚举类型是使用enum关键字声明的。

```
enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, S
enum Months : byte { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, O
```

默认情况下，枚举中每个元素的基础类型是 int。可以使用冒号指定另一种整数值类型，如前面的示例所示。有关可能的类型的完整列表，请参见 enum（C# 参考）。

可以通过转换验证基础数值与基础类型，如下例所示。

```
Days today = Days.Monday;
int dayNumber =(int)today;
Console.WriteLine("{0} is day number #{1}.", today, dayNumber);

Months thisMonth = Months.Dec;
byte monthNumber = (byte)thisMonth;
Console.WriteLine("{0} is month number #{1}.", thisMonth, monthNumb

// Output:
// Monday is day number #1.
// Dec is month number #11.
```

以下是使用枚举而不使用数值类型的好处：

- 明确为客户端代码指定哪些值是变量的有效值。
- 在 Visual Studio 中，IntelliSense 列出定义的值。

如果未在枚举数列表中指定元素的值，则值将自动按 1 递增。在前面的示例中，Days.Sunday 的值为 0，Days.Monday 的值为 1，依此类推。创建新的 Days 对象时，如果不显式为其赋值，则它将具有默认值 Days.Sunday (0)。创建枚举时，应选择最合理的默认值并赋给它一个零值。这便使得只要在创建枚举时未为其显式赋值，则所创建的全部枚举都将具有该默认值。

如果变量 meetingDay 的类型为 Days，则只能将 Days 定义的某个值赋给它（无需显式强制转换）。如果会议日期更改，可以将 Days 中的新值赋给 meetingDay：

```
Days meetingDay = Days.Monday;
//...
meetingDay = Days.Friday;
```

> 注意
>
> 可以将任意整数值赋给 meetingDay。例如，代码行 meetingDay = (Days) 42 不会产生错误。但也不应该这样做，因为默认约定的是枚举变量只容纳枚举定义的值之一。将任意值赋给枚举类型的变量很有可能会导致错误。

可以将任意值赋给枚举类型的枚举数列表中的元素，也可以使用计算值：

```
enum MachineState
{
    PowerOff = 0,
    Running = 5,
    Sleeping = 10,
    Hibernating = Sleeping + 5
}
```

枚举类型作为位标志

可以使用枚举类型定义位标志，从而使该枚举类型的实例可以存储枚举数列表中定义的值的任意组合。（当然，某些组合在您的程序代码中可能没有意义或不允许使用。）

创建位标志枚举的方法是应用 System.FlagsAttribute 特性并适当定义一些值，以便可以对这些值执行 **AND**、**OR**、**NOT** 和 **XOR** 按位运算。在位标志枚举中包含一个值为零（表示“未设置任何标志”）的命名常量。如果零值不表示“未设置任何标志”，则请不要为标志指定零值。

在下面的示例中，定义了 Days 枚举的另一个版本，命名为 Days2。 Days2 具有 Flags 特性，且它的每个值都是 2 的若干次幂，指数依次递增。这样，您将能够创建值为 Days2.Tuesday 和 Days2.Thursday 的 Days2 变量。

```
[Flags]
enum Days2
{
    None = 0x0,
    Sunday = 0x1,
    Monday = 0x2,
    Tuesday = 0x4,
    Wednesday = 0x8,
    Thursday = 0x10,
    Friday = 0x20,
    Saturday = 0x40
}
class MyClass
{
    Days2 meetingDays = Days2.Tuesday | Days2.Thursday;
}
```

若要在某个枚举上设置标志，请使用按位 **OR** 运算符，如下面的示例所示：

```
// Initialize with two flags using bitwise OR.
meetingDays = Days2.Tuesday | Days2.Thursday;

// Set an additional flag using bitwise OR.
meetingDays = meetingDays | Days2.Friday;

Console.WriteLine("Meeting days are {0}", meetingDays);
// Output: Meeting days are Tuesday, Thursday, Friday

// Remove a flag using bitwise XOR.
meetingDays = meetingDays ^ Days2.Tuesday;
Console.WriteLine("Meeting days are {0}", meetingDays);
// Output: Meeting days are Thursday, Friday
```

若要确定是否设置了特定标志，请使用按位 **AND** 运算，如下面的示例所示：

```
// Test value of flags using bitwise AND.
bool test = (meetingDays & Days2.Thursday) == Days2.Thursday;
Console.WriteLine("Thursday {0} a meeting day.", test == true ? "is
// Output: Thursday is a meeting day.
```

有关使用 System.FlagsAttribute 特性定义枚举类型时需要考虑的事项的更多信息，请参见 System.Enum。

使用 System.Enum 方法发现和操作枚举值

所有枚举都是 System.Enum 类型的实例。不能从 System.Enum 派生新类，但可以使用它的方法发现有关枚举实例中的值的信息以及操作这些值。

```
string s = Enum.GetName(typeof(Days), 4);
Console.WriteLine(s);

Console.WriteLine("The values of the Days Enum are:");
foreach (int i in Enum.GetValues(typeof(Days)))
    Console.WriteLine(i);

Console.WriteLine("The names of the Days Enum are:");
foreach (string str in Enum.GetNames(typeof(Days)))
    Console.WriteLine(str);
```

有关更多信息，请参见AllMembers.T:System.Enum。

此外，还可以使用扩展方法为枚举创建新方法。有关更多信息，请参见如何：为枚举创建新方法（C# 编程指南）。

重要章节

更多有关变量 在 开始 Visual c# 2010

请参阅

System.Enum

C# 编程指南

enum（C# 参考）

事件（C# 编程指南）

类或对象可以通过事件向其他类或对象通知发生的相关事情。发送（或引发）事件的类称为“发行者”，接收（或处理）事件的类称为“订户”。

在典型的 C# Windows 窗体或 Web 应用程序中，可订阅由控件（如按钮和列表框）引发的事件。可使用 Visual C# 集成开发环境 (IDE) 来浏览控件发布的事件，选择要处理的事件。IDE 会自动添加空事件处理程序方法和订阅事件的代码。有关更多信息，请参见 如何：订阅和取消订阅事件（C# 编程指南）。

事件概述

事件具有以下特点：

- 发行者确定何时引发事件，订户确定执行何种操作来响应该事件。
- 一个事件可以有多个订户。一个订户可处理来自多个发行者的多个事件。
- 没有订户的事件永远也不会引发。
- 事件通常用于通知用户操作，例如，图形用户界面中的按钮单击或菜单选择操作。
- 如果一个事件有多个订户，当引发该事件时，会同步调用多个事件处理程序。要异步调用事件，请参见Calling Synchronous Methods Asynchronously。
- 在 .NET Framework 类库中，事件是基于 EventHandler 委托和 EventArgs 基类的。

相关章节

有关更多信息，请参见：

- 如何：订阅和取消订阅事件（C# 编程指南）
- 如何：发布符合 .NET Framework 准则的事件（C# 编程指南）
- 如何：在派生类中引发基类事件（C# 编程指南）
- 如何：实现接口事件（C# 编程指南）
- 线程同步（C# 和 Visual Basic）
- 如何：使用字典存储事件实例（C# 编程指南）
- 如何：实现自定义事件访问器（C# 编程指南）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

重要章节

http://go.microsoft.com/fwlink/?LinkId=195395

http://go.microsoft.com/fwlink/?LinkId=195369

http://go.microsoft.com/fwlink/?LinkId=195418

http://go.microsoft.com/fwlink/?LinkId=195412

请参阅

EventHandler

C# 编程指南

委托（C# 编程指南）

在 Windows 窗体中创建事件处理程序

Multithreaded Programming with the Event-based Asynchronous Pattern

如何：订阅和取消订阅事件（C# 编程指南）

如果您想编写引发事件时调用的自定义代码，则可以订阅由其他类发布的事件。例如，可以订阅某个按钮的 click 事件，以使应用程序在用户单击该按钮时执行一些有用的操作。

使用 Visual Studio IDE 订阅事件

1. 如果看不到“属性”窗口，请在“设计”视图中，右击要为其创建事件处理程序的窗体或控件，然后选择“属性”。
2. 在“属性”窗口的顶部，单击“事件”图标。
3. 双击要创建的事件，例如 Load 事件。

   Visual C# 会创建一个空事件处理程序方法，并将其添加到您的代码中。或者，您也可以在“代码”视图中手动添加代码。例如，下面的代码行声明了一个在 Form 类引发 Load 事件时调用的事件处理程序方法。

   ```
   private void Form1_Load(object sender, System.EventArgs e)
   {
       // Add your form load event handling code here.
   }
   ```

   还会在项目的 Form1.Designer.cs 文件的 InitializeComponent 方法中自动生成订阅该事件所需的代码行。该代码行类似于：

   ```
   this.Load += new System.EventHandler(this.Form1_Load);
   ```

以编程方式订阅事件

1. 定义一个事件处理程序方法，其签名与该事件的委托签名匹配。例如，如果事件基于 EventHandler 委托类型，则下面的代码表示方法存根：

   ```
   void HandleCustomEvent(object sender, CustomEventArgs a)
   {
      // Do something useful here.
   }
   ```

2. 使用加法赋值运算符 (+=) 来为事件附加事件处理程序。在下面的示例中，假设名为 publisher 的对象拥有一个名为 RaiseCustomEvent 的事件。请注意，订户类需要引用发行者类才能订阅其事件。

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

请注意，前面的语法是 C# 2.0 中的新语法。此语法完全等效于必须使用 **new** 关键字显式创建封装委托的 C# 1.0 语法：

```
publisher.RaiseCustomEvent += new CustomEventHandler(HandleCust
```

还可以通过使用 lambda 表达式添加事件处理程序：

```
public Form1()
{
    InitializeComponent();
    // Use a lambda expression to define an event handler.
    this.Click += (s,e) =&gt; { MessageBox.Show(
       ((MouseEventArgs)e).Location.ToString());};
}
```

有关更多信息，请参见 如何：在 LINQ 外部使用 Lambda 表达式（C# 编程指南）。

使用匿名方法订阅事件

- 如果以后不必取消订阅某个事件，则可以使用加法赋值运算符 (**+=**) 将匿名方法附加到此事件。在下面的示例中，假设名为 publisher 的对象拥有一个名为 RaiseCustomEvent 的事件，并且还定义了一个 CustomEventArgs 类以承载某些类型的专用事件信息。请注意，订户类需要引用 publisher 才能订阅其事件。

```
publisher.RaiseCustomEvent += delegate(object o, CustomEventArg
{
  string s = o.ToString() + " " + e.ToString();
  Console.WriteLine(s);
};
```

请务必注意，如果使用匿名函数订阅事件，事件的取消订阅过程将比较麻烦。这种情况下若要取消订阅，必须返回到该事件的订阅代码，将该匿名方法存储在委托变量中，然后将此委托添加到该事件中。一般来说，如果必须在后面的代码中取消订阅某个事件，则建议您不要使用匿名函数订阅此事件。有关匿名函数的更多信息，请参见匿名函数（C# 编程指南）。

取消订阅

要防止在引发事件时调用事件处理程序，请取消订阅该事件。要防止资源泄露，应在释放订户对象之前取消订阅事件。在取消订阅事件之前，在发布对象中作为该事件的基础的多路广播委托会引用封装了订户的事件处理程序的委托。只要发布对象保持该引用，垃圾回收功能就不会删除订户对象。

取消订阅事件

- 使用减法赋值运算符 (**-=**) 取消订阅事件：

  ```
  publisher.RaiseCustomEvent -= HandleCustomEvent;
  ```

  所有订户都取消订阅事件后，发行者类中的事件实例将设置为 **null**。

请参阅

事件（C# 编程指南）

event（C# 参考）

如何：发布符合 .NET Framework 准则的事件（C# 编程指南）

-= 运算符（C# 参考）

+= 运算符（C# 参考）

如何：发布符合 .NET Framework 准则的事件（C# 编程指南）

下面的过程演示了如何将符合标准 .NET Framework 模式的事件添加到您的类和结构中。.NET Framework 类库中的所有事件均基于 EventHandler 委托，定义如下：

```
public delegate void EventHandler(object sender, EventArgs e);
```

> 注意
>
> .NET Framework 2.0 引入了此委托的一个泛型版本，即 EventHandler<TEventArgs>。下面的示例显示如何使用这两种版本。

虽然您定义的类中的事件可基于任何有效委托类型（甚至是可返回值的委托），但是，通常建议您使用 EventHandler 让事件基于 .NET Framework 模式，如下面的示例所示。

采用 EventHandler 模式发布事件

1. （如果不需要与事件一起发送自定义数据，请跳过此步骤，进入步骤 3a。）在发行者类和订阅方类均可看见的范围中声明自定义数据的类。然后添加保留您的自定义事件数据所需的成员。在此示例中，会返回一个简单字符串。

```
public class CustomEventArgs : EventArgs
{
    public CustomEventArgs(string s)
    {
        msg = s;
    }
    private string msg;
    public string Message
    {
        get { return msg; }
    }
}
```

2. （如果您使用的是 EventHandler<TEventArgs> 的泛型版本，请跳过此步骤。）在发布类中声明一个委托。为它指定以 EventHandler 结尾的名称。第二个参数指定自定义 EventArgs 类型。

```
public delegate void CustomEventHandler(object sender, CustomEv
```

3. 使用以下任一步骤，在发布类中声明事件。

   i. 如果没有自定义 EventArgs 类，事件类型就是非泛型 EventHandler 委托。无需声明委托，因为它已在创建 C# 项目时包含的 System 命名空间中进行了声明。将以下代码添加到发行者类中。

   ```
   public event EventHandler RaiseCustomEvent;
   ```

   ii. 如果使用的是 EventHandler 的非泛型版本，并且您有一个由 EventArgs 派生的自定义类，请在发布类中声明您的事件，并且将来自步骤 2 的委托用作类型。

   ```
   public event CustomEventHandler RaiseCustomEvent;
   ```

   iii. 如果使用的是泛型版本，则不需要自定义委托。相反，在发行者类中，您应将事件类型指定为 EventHandler<CustomEventArgs>，将尖括号中的内容替换为自己的类的名称。

   ```
   public event EventHandler&lt;CustomEventArgs&gt; RaiseCusto
   ```

下面的示例通过将自定义 EventArgs 类和 EventHandler<TEventArgs> 用作事件类型来演示上述步骤。

```
namespace DotNetEvents
{
    using System;
    using System.Collections.Generic;

    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string s)
        {
            message = s;
        }
        private string message;

        public string Message
        {
            get { return message; }
            set { message = value; }
```

```
        }
    }

    // Class that publishes an event
    class Publisher
    {

        // Declare the event using EventHandler<T>
        public event EventHandler<CustomEventArgs> RaiseCustomEvent

        public void DoSomething()
        {
            // Write some code that does something useful here
            // then raise the event. You can also raise an event
            // before you execute a block of code.
            OnRaiseCustomEvent(new CustomEventArgs("Did something")

        }

        // Wrap event invocations inside a protected virtual method
        // to allow derived classes to override the event invocatio
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e
        {
            // Make a temporary copy of the event to avoid possibil
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the even
            EventHandler<CustomEventArgs> handler = RaiseCustomEven

            // Event will be null if there are no subscribers
            if (handler != null)
            {
                // Format the string to send inside the CustomEvent
                e.Message += String.Format(" at {0}", DateTime.Now.

                // Use the () operator to raise the event.
                handler(this, e);
            }
        }
    }

    //Class that subscribes to an event
    class Subscriber
    {
        private string id;
        public Subscriber(string ID, Publisher pub)
        {
            id = ID;
            // Subscribe to the event using C# 2.0 syntax
            pub.RaiseCustomEvent += HandleCustomEvent;
        }

        // Define what actions to take when the event is raised.
        void HandleCustomEvent(object sender, CustomEventArgs e)
```

```
            {
                Console.WriteLine(id + " received this message: {0}", e
            }
        }

        class Program
        {
            static void Main(string[] args)
            {
                Publisher pub = new Publisher();
                Subscriber sub1 = new Subscriber("sub1", pub);
                Subscriber sub2 = new Subscriber("sub2", pub);

                // Call the method that raises the event.
                pub.DoSomething();

                // Keep the console window open
                Console.WriteLine("Press Enter to close this window.");
                Console.ReadLine();

            }
        }
    }
```

请参阅

Delegate

C# 编程指南

事件（C# 编程指南）

委托（C# 编程指南）

如何：在派生类中引发基类事件（C# 编程指南）

以下简单示例演示了在基类中声明可从派生类引发的事件的标准方法。此模式广泛应用于 .NET Framework 类库中的 Windows 窗体类。

在创建可用作其他类的基类的类时，应考虑如下事实：事件是特殊类型的委托，只可以从声明它们的类中调用。派生类无法直接调用基类中声明的事件。尽管有时需要事件仅由基类引发，但在大多数情形下，应该允许派生类调用基类事件。为此，您可以在包含该事件的基类中创建一个受保护的调用方法。通过调用或重写此调用方法，派生类便可以间接调用该事件。

> 注意
>
> 不要在基类中声明虚拟事件，也不要在派生类中重写这些事件。C# 编译器无法正确处理这些事件，并且无法预知的该派生的事件的用户是否真正订阅了基类事件。

```
namespace BaseClassEvents
{
    using System;
    using System.Collections.Generic;

    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        private double newArea;

        public ShapeEventArgs(double d)
        {
            newArea = d;
        }
        public double NewArea
        {
            get { return newArea; }
        }
    }

    // Base class event publisher
    public abstract class Shape
    {
        protected double area;

        public double Area
        {
            get { return area; }
            set { area = value; }
        }
```

```
        // The event. Note that by using the generic EventHandler<T
        // we do not need to declare a separate delegate type.
        public event EventHandler<ShapeEventArgs> ShapeChanged;

        public abstract void Draw();

        //The event-invoking method that derived classes can overri
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
            // Make a temporary copy of the event to avoid possibil
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the even
            EventHandler<ShapeEventArgs> handler = ShapeChanged;
            if (handler != null)
            {
                handler(this, e);
            }
        }
    }

    public class Circle : Shape
    {
        private double radius;
        public Circle(double d)
        {
            radius = d;
            area = 3.14 * radius * radius;
        }
        public void Update(double d)
        {
            radius = d;
            area = 3.14 * radius * radius;
            OnShapeChanged(new ShapeEventArgs(area));
        }
        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any circle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }
        public override void Draw()
        {
            Console.WriteLine("Drawing a circle");
        }
    }

    public class Rectangle : Shape
    {
        private double length;
        private double width;
        public Rectangle(double length, double width)
        {
```

```
            this.length = length;
            this.width = width;
            area = length * width;
        }
        public void Update(double length, double width)
        {
            this.length = length;
            this.width = width;
            area = length * width;
            OnShapeChanged(new ShapeEventArgs(area));
        }
        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any rectangle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }
        public override void Draw()
        {
            Console.WriteLine("Drawing a rectangle");
        }

    }

    // Represents the surface on which the shapes are drawn
    // Subscribes to shape events so that it knows
    // when to redraw a shape.
    public class ShapeContainer
    {
        List<Shape> _list;

        public ShapeContainer()
        {
            _list = new List<Shape>();
        }

        public void AddShape(Shape s)
        {
            _list.Add(s);
            // Subscribe to the base class event.
            s.ShapeChanged += HandleShapeChanged;
        }

        // ...Other methods to draw, resize, etc.

        private void HandleShapeChanged(object sender, ShapeEventAr
        {
            Shape s = (Shape)sender;

            // Diagnostic message for demonstration purposes.
            Console.WriteLine("Received event. Shape area is now {0
```

```
                // Redraw the shape here.
                s.Draw();
            }
        }

    class Test
    {

        static void Main(string[] args)
        {
            //Create the event publishers and subscriber
            Circle c1 = new Circle(54);
            Rectangle r1 = new Rectangle(12, 9);
            ShapeContainer sc = new ShapeContainer();

            // Add the shapes to the container.
            sc.AddShape(c1);
            sc.AddShape(r1);

            // Cause some events to be raised.
            c1.Update(57);
            r1.Update(7, 7);

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
/* Output:
        Received event. Shape area is now 10201.86
        Drawing a circle
        Received event. Shape area is now 49
        Drawing a rectangle
 */
```

请参阅

C# 编程指南

事件（C# 编程指南）

委托（C# 编程指南）

访问修饰符（C# 编程指南）

在 Windows 窗体中创建事件处理程序

如何：实现接口事件（C# 编程指南）

接口可声明事件。下面的示例演示如何在类中实现接口事件。实现接口事件的规则与实现任何接口方法或属性的规则基本相同。

在类中实现接口事件

- 在类中声明事件，然后在适当的区域调用该事件。

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event…

            OnShapeChanged(new MyEventArgs(/*arguments*/));

            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            if(ShapeChanged != null)
            {
               ShapeChanged(this, e);
            }
        }
    }

}
```

下面的示例演示如何处理以下的不常见情况：您的类是从两个以上的接口继承的，每个接口都含有同名事件）。在这种情况下，您至少要为其中一个事件提供显式接口实现。为事件编写显式接口实现时，必须编写 **add** 和 **remove** 事件访问器。这两个事件访问器通常由编译器提供，但在这种情况下编译器不能提供。

您可以提供自己的访问器，以便指定这两个事件是由您的类中的同一事件表示，还是由不同事件表示。例如，根据接口规范，如果事件应在不同时间引发，则可以将每个事件与类中的一个单独实现关联。在下面的示例中，订户将形状引用强制转换为 IShape 或 IDrawingObject，从而确定自己将会接收哪个 OnDraw 事件。

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }
    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }

    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
        event EventHandler IDrawingObject.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PreDrawEvent -= value;
                }
            }
        }
```

```
        // Explicit interface implementation required.
        // Associate IShape's event with
        // PostDrawEvent
        event EventHandler IShape.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PostDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PostDrawEvent -= value;
                }
            }

        }

        // For the sake of simplicity this one method
        // implements both interfaces.
        public void Draw()
        {
            // Raise IDrawingObject's event before the object is dr
            EventHandler handler = PreDrawEvent;
            if (handler != null)
            {
                handler(this, new EventArgs());
            }
            Console.WriteLine("Drawing a shape.");

            // RaiseIShape's event after the object is drawn.
            handler = PostDrawEvent;
            if (handler != null)
            {
                handler(this, new EventArgs());
            }
        }
    }
    public class Subscriber1
    {
        // References the shape object as an IDrawingObject
        public Subscriber1(Shape shape)
        {
            IDrawingObject d = (IDrawingObject)shape;
            d.OnDraw += new EventHandler(d_OnDraw);
        }

        void d_OnDraw(object sender, EventArgs e)
        {
```

```
                Console.WriteLine("Sub1 receives the IDrawingObject eve
            }
        }
        // References the shape object as an IShape
        public class Subscriber2
        {
            public Subscriber2(Shape shape)
            {
                IShape d = (IShape)shape;
                d.OnDraw += new EventHandler(d_OnDraw);
            }

            void d_OnDraw(object sender, EventArgs e)
            {
                Console.WriteLine("Sub2 receives the IShape event.");
            }
        }

        public class Program
        {
            static void Main(string[] args)
            {
                Shape shape = new Shape();
                Subscriber1 sub = new Subscriber1(shape);
                Subscriber2 sub2 = new Subscriber2(shape);
                shape.Draw();

                // Keep the console window open in debug mode.
                System.Console.WriteLine("Press any key to exit.");
                System.Console.ReadKey();
            }
        }

    }
    /* Output:
        Sub1 receives the IDrawingObject event.
        Drawing a shape.
        Sub2 receives the IShape event.
    */
```

请参阅

C# 编程指南

事件（C# 编程指南）

委托（C# 编程指南）

显式接口实现（C# 编程指南）

如何：在派生类中引发基类事件（C# 编程指南）

如何：使用字典存储事件实例（C# 编程指南）

accessor-declarations 的一种用法是公开很多事件但不为每个事件分配字段，而是使用字典来存储这些事件实例。这只在具有很多事件但您预计大多数事件都不会实现时才有用。

```
public delegate void EventHandler1(int i);
public delegate void EventHandler2(string s);

public class PropertyEventsSample
{
    private System.Collections.Generic.Dictionary<string, System.De

    public PropertyEventsSample()
    {
        eventTable = new System.Collections.Generic.Dictionary<stri
        eventTable.Add("Event1", null);
        eventTable.Add("Event2", null);
    }

    public event EventHandler1 Event1
    {
        add
        {
            lock (eventTable)
            {
                eventTable["Event1"] = (EventHandler1)eventTable["E
            }
        }
        remove
        {
            lock (eventTable)
            {
                eventTable["Event1"] = (EventHandler1)eventTable["E
            }
        }
    }

    public event EventHandler2 Event2
    {
        add
        {
            lock (eventTable)
            {
                eventTable["Event2"] = (EventHandler2)eventTable["E
            }
        }
        remove
        {
```

```
                lock (eventTable)
                {
                    eventTable["Event2"] = (EventHandler2)eventTable["E
                }
            }
        }

        internal void RaiseEvent1(int i)
        {
            EventHandler1 handler1;
            if (null != (handler1 = (EventHandler1)eventTable["Event1"]
            {
                handler1(i);
            }
        }

        internal void RaiseEvent2(string s)
        {
            EventHandler2 handler2;
            if (null != (handler2 = (EventHandler2)eventTable["Event2"]
            {
                handler2(s);
            }
        }
    }

    public class TestClass
    {
        public static void Delegate1Method(int i)
        {
            System.Console.WriteLine(i);
        }

        public static void Delegate2Method(string s)
        {
            System.Console.WriteLine(s);
        }

        static void Main()
        {
            PropertyEventsSample p = new PropertyEventsSample();

            p.Event1 += new EventHandler1(TestClass.Delegate1Method);
            p.Event1 += new EventHandler1(TestClass.Delegate1Method);
            p.Event1 -= new EventHandler1(TestClass.Delegate1Method);
            p.RaiseEvent1(2);

            p.Event2 += new EventHandler2(TestClass.Delegate2Method);
            p.Event2 += new EventHandler2(TestClass.Delegate2Method);
            p.Event2 -= new EventHandler2(TestClass.Delegate2Method);
            p.RaiseEvent2("TestString");

            // Keep the console window open in debug mode.
```

```
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
    /* Output:
        2
        TestString
    */
```

请参阅

C# 编程指南

事件（C# 编程指南）

委托（C# 编程指南）

如何：实现自定义事件访问器（C# 编程指南）

事件是特殊类型的多路广播委托，只能从声明它的类中调用。客户端代码通过提供对应在引发事件时调用的方法的引用来订阅事件。这些方法通过事件访问器添加到委托的调用列表中，事件访问器类似于属性访问器，不同之处在于事件访问器被命名为 **add** 和 **remove**。在大多数情况下都不需要提供自定义的事件访问器。如果您在代码中没有提供自定义的事件访问器，编译器会自动添加事件访问器。但在某些情况下，您可能需要提供自定义行为。在主题如何：实现接口事件（C# 编程指南）中就演示了这样一种情况。

下面的示例演示如何实现自定义的 add 和 remove 事件访问器。虽然可以替换这些访问器内的任何代码，但建议您在添加或移除新的事件处理程序方法之前先锁定该事件。

```
event EventHandler IDrawingObject.OnDraw
        {
            add
            {
                lock (PreDrawEvent)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
                lock (PreDrawEvent)
                {
                    PreDrawEvent -= value;
                }
            }
        }
```

请参阅

事件（C# 编程指南）

event（C# 参考）

异常和异常处理（C# 编程指南）

C# 语言的异常处理功能可帮助您处理程序运行时出现的任何意外或异常情况。异常处理使用 **try**、**catch** 和 **finally** 关键字尝试某些操作，以处理失败情况，尽管这些操作有可能失败，但如果您确定需要这样做，且希望在事后清理资源，就可以尝试这样做。公共语言运行时 (CLR)、.NET Framework 或任何第三方库或者应用程序代码都可以生成异常。异常是使用 **throw** 关键字创建的。

很多情况下，异常可能不是由代码直接调用的方法引发，而是由调用堆栈中位置更靠下的另一个方法所引发。在这种情况下，CLR 将展开堆栈，查找是否有方法包含针对该特定异常类型的 **catch** 块，如果找到这样的方法，就会执行找到的第一个这样的 **catch** 块。如果在调用堆栈中的任何位置都没有找到适当的 **catch** 块，就会终止该进程，并向用户显示一条消息。

此示例中使用一个方法检测是否有被零除的情况；如果有，则捕获该错误。如果没有异常处理，此程序将终止并产生**“DivideByZeroException** 未处理**”**错误。

```
class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new System.DivideByZeroException();
        return x / y;
    }
    static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result = 0;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, res
        }
        catch (DivideByZeroException e)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

异常概述

异常具有以下特点：

- 各种类型的异常最终都是由 **System.Exception** 派生而来。
- 在可能引发异常的语句周围使用 **try** 块。
- 一旦 **try** 块中发生异常，控制流将跳转到第一个关联的异常处理程序（无论该处理程序存在于调用堆栈中的什么位置）。在 C# 中，**catch** 关键字用于定义异常处理程序。
- 如果给定异常没有异常处理程序，则程序将停止执行，并显示一条错误消息。
- 除非您可以处理某个异常并使应用程序处于已知状态，否则请不要捕捉该异常。如果捕捉 **System.Exception**，请在 **catch** 块的末尾使用 **throw** 关键字再次引发该异常。
- 如果 **catch** 块定义了一个异常变量，则可以用它获取有关所发生异常类型的更多信息。
- 程序可以使用 **throw** 关键字显式地引发异常。
- 异常对象包含有关错误的详细信息，比如调用堆栈的状态以及有关错误的文本说明。
- 即使发生异常也会执行 **finally** 块中的代码。使用 **finally** 块释放资源，例如，关闭在 **try** 块中打开的任何流或文件。
- .NET Framework 中的托管异常是凭借 Win32 结构化异常处理机制实现的。有关更多信息，请参见结构化异常处理 (C/C++) 和 A Crash Course on the Depths of Win32 Structured Exception Handling（有关深入探究 Win32 结构化异常处理的应急课程）。

相关章节

有关异常和异常处理的更多信息，请参见以下主题：

- 使用异常（C# 编程指南）
- 异常处理（C# 编程指南）
- 创建和引发异常（C# 编程指南）
- 编译器生成的异常（C# 编程指南）
- 如何：使用 try/catch 处理异常（C# 编程指南）
- 如何：使用 finally 执行清理代码（C# 编程指南）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

SystemException

C# 编程指南

C# 关键字

throw（C# 参考）

try-catch（C# 参考）

try-finally（C# 参考）

try-catch-finally（C# 参考）

处理和引发异常

异常层次结构

编写可靠的 .NET 代码

特定异常的 Minidumps

使用异常（C# 编程指南）

在 C# 中，程序中的运行时错误通过使用一种称为“异常”的机制在程序中传播。 异常由遇到错误的代码引发，由能够更正错误的代码捕捉。 异常可由 .NET Framework 公共语言运行时 (CLR) 或由程序中的代码引发。 一旦引发了一个异常，这个异常就会在调用堆栈中往上传播，直到找到针对它的 catch 语句。 未捕获的异常由系统提供的通用异常处理程序处理，该处理程序会显示一个对话框。

异常由从 Exception 派生的类表示。 此类标识异常的类型，并包含详细描述异常的属性。 引发异常涉及到创建一个异常派生类的实例，配置异常的属性（可选），然后使用 throw 关键字引发该对象。 例如：

```
class CustomException : Exception
{
    public CustomException(string message)
    {

    }

}
private static void TestThrow()
{
    CustomException ex =
        new CustomException("Custom exception in TestThrow()");

    throw ex;
}
```

在引发异常之后，运行时检查当前语句以确定它是否在 try 块中。 如果是，则检查与该 try 块关联的任何 catch 块，以确定它们是否能够捕获该异常。 Catch 块通常会指定异常类型；如果该 catch 块的类型与异常或异常的基类的类型相同，则该 catch 块就能够处理该方法。 例如：

```
static void TestCatch()
{
    try
    {
        TestThrow();
    }
    catch (CustomException ex)
    {
        System.Console.WriteLine(ex.ToString());
    }
}
```

如果引发异常的语句不在 try 块中，或者包含该语句的 try 块没有匹配的 catch 块，运行时将检查调用方法中是否有 try 语句和 catch 块。 运行时将在调用堆栈中向上继续搜索兼容的 catch 块。 在找到并执行 catch 块之后，控制权将传递给 catch 块之后的下一个语句。

一个 try 语句可能包含多个 catch 块。 将执行第一个能够处理该异常的 catch 语句；任何后续的 catch 语句都将被忽略，即使它们是兼容的也如此。 因此，在任何情况下都应该按照从最具体（或者派生程度最高）到最不具体这一顺序排列 catch 块。 例如：

```
static void TestCatch2()
{
    System.IO.StreamWriter sw = null;
    try
    {
        sw = new System.IO.StreamWriter(@"C:\test\test.txt");
        sw.WriteLine("Hello");
    }

    catch (System.IO.FileNotFoundException ex)
    {
        // Put the more specific exception first.
        System.Console.WriteLine(ex.ToString());
    }

    catch (System.IO.IOException ex)
    {
        // Put the less specific exception last.
        System.Console.WriteLine(ex.ToString());
    }
    finally
    {
        sw.Close();
    }

    System.Console.WriteLine("Done");
}
```

执行 catch 块之前，运行时会检查 finally 块。 Finally 块使程序员能够清除中止的 try 块可能遗留下的任何模糊状态，或者释放任何外部资源（例如图形句柄、数据库连接或文件流），而无需等待运行时中的垃圾回收器终结这些对象。 例如：

```
static void TestFinally()
{
    System.IO.FileStream file = null;
    //Change the path to something that works on your machine.
    System.IO.FileInfo fileInfo = new System.IO.FileInfo(@"C:\file

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - o
        if (file != null)
        {
            file.Close();
        }
    }

    try
    {
        file = fileInfo.OpenWrite();
        System.Console.WriteLine("OpenWrite() succeeded");
    }
    catch (System.IO.IOException)
    {
        System.Console.WriteLine("OpenWrite() failed");
    }
}
```

如果 WriteByte() 引发了异常，那么在没有调用 file.Close() 的情况下，第二个 try 块中尝试重新打开文件的代码就会失败，并且文件将保持锁定状态。 由于要执行 finally 块（即使已引发异常），前一示例中的 finally 块使得可以正确地关闭文件，从而帮助避免错误。

如果在引发异常之后没有在调用堆栈上找到兼容的 catch 块，则会出现三种情况中的一种：

- 如果异常出现在析构函数中，则中止该析构函数并调用基析构函数（如果有）。
- 如果调用堆栈包含静态构造函数或静态字段初始值设定项，则引发一个 TypeInitializationException，并将原始异常分配给新异常的 InnerException 属性。
- 如果到达线程的开头，则终止线程。

请参见

参考

异常和异常处理（C# 编程指南）

概念

C# 编程指南

异常处理（C# 编程指南）

C# 程序员可使用 try 块对可能受异常影响的代码进行分区。关联的 catch 块用于处理任何结果异常。一个包含代码的 finally 块，无论 **try** 块中是否引发异常（例如，释放在 **try** 块中分配的资源），这些代码都会运行。一个 **try** 块需要一个或多个关联的 **catch** 块或一个 **finally** 块，或两者。

以下示例给出了一个 **try-catch** 语句，一个 **try-finally** 语句，和一个 **try-catch-finally** 语句。

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

不带有 **catch** 或 **finally** 块的 **try** 块将导致编译器错误。

Catch 块

catch 块可以指定要捕捉的异常的该类型。类型规范称为“异常筛选器”。异常类型应从 Exception 派生出来。一般而言，不会将 Exception 指定为异常筛选器，除非您了解如何处理 **try** 块中可能引发的所有异常，或者您在 **catch** 块中包括了 throw 语句。

具有不同异常筛选器的多个 **catch** 块可以串联在一起。多个 **catch** 数据块的计算顺序是在代码中从顶部到底部，但是，对于所引发的每个异常，都只执行一个 **catch** 数据块。与指定的准确类型或其基类最为匹配的第一个 **catch** 块被执行。如果 **catch** 块没有指定匹配异常筛选器，则 **catch** 块就不具有选定的筛选器（如果语句有的话）。需要将带有最具体的（即派生程度最高的）异常类的 **catch** 块放在最前面。

当下列条件为真时，应该捕捉异常：

- 对引发异常的原因有具体的了解，并可实现特定的恢复，例如，在捕获 FileNotFoundException 对象时提示用户输入新的文件名。
- 可以新建一个更具体的异常并引发该异常。

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch(System.IndexOutOfRangeException e)
    {
        throw new System.ArgumentOutOfRangeException(
            "Parameter index is out of range.");
    }
}
```

- 希望在将异常传递出去进行额外处理前部分地处理异常。在下面的示例中，**catch** 块用于在再次引发异常之前，向错误日志添加条目。

```
try
{
    // Try to access a resource.
}
catch (System.UnauthorizedAccessException e)
{
    // Call a custom error logging procedure.
    LogError(e);
    // Re-throw the error.
    throw;
}
```

Finally 块

可以使用 **finally** 块清理在 **try** 块中执行的操作。如果存在，**finally** 块将在最后执行，在 **try** 块和任何匹配 **catch** 的块之后执行。不管是否引发异常或者是否找到与异常类型匹配的 **catch** 块，**finally** 始终运行。

可以使用 **finally** 块释放资源（如文件流、数据库连接和图形句柄），而不用等待由运行时中的垃圾回收器来完成对象。有关更多信息，请参见using 语句（C# 参考）。

在下面的示例中，使用 **finally** 块关闭在 **try** 块中打开的文件。注意，在关闭文件之前要检查该文件句柄的状态。如果 **try** 块无法打开文件，则文件句柄仍具有值 **null**，并且 **finally** 块不会尝试关闭它。或者，如果在 **try** 块中成功打开该文件，则 **finally** 块将关闭打开的文件。

```
System.IO.FileStream file = null;
System.IO.FileInfo fileinfo = new System.IO.FileInfo("C:\\file.txt'
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    if (file != null)
    {
        file.Close();
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

异常和异常处理（C# 编程指南）

try-catch（C# 参考）

try-finally（C# 参考）

try-catch-finally（C# 参考）

using 语句（C# 参考）

创建和引发异常（C# 编程指南）

异常用于指示在运行程序时发生了错误。此时将创建一个描述错误的异常对象，然后使用 throw 关键字“引发”该对象。然后运行时搜索最兼容的异常处理程序。

当存在下列一种或多种情况时，程序员应引发异常：

- 方法无法完成其中定义的功能。

  例如，如果方法的参数具有无效值：

```
static void CopyObject(SampleClass original)
{
    if (original == null)
    {
        throw new System.ArgumentException("Parameter cannot be
    }

}
```

- 根据对象的状态，对某个对象进行不适当的调用。

  一个示例可能尝试对只读文件执行写操作。在对象状态不允许某项操作的情况下，引发 InvalidOperationException 的一个实例或基于此类的派生类的对象。以下为引发 InvalidOperationException 对象的方法的示例：

```
class ProgramLog
{
    System.IO.FileStream logFile = null;
    void OpenLog(System.IO.FileInfo fileName, System.IO.FileMod

    void WriteLog()
    {
        if (!this.logFile.CanWrite)
        {
            throw new System.InvalidOperationException("Logfile
        }
        // Else write data to the log and return.
    }
}
```

- 方法的参数导致了异常。

  在此情况下，应捕获原始异常并创建一个 ArgumentException 实例。原始异常应作为 InnerException 参数传递给 ArgumentException 的构造函数：

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        System.ArgumentException argEx = new System.ArgumentExc
        throw argEx;
    }
}
```

异常包含一个名为 StackTrace 的属性。此字符串包含当前调用堆栈上的方法的名称，以及为每个方法引发异常的位置（文件名和行号）。 StackTrace 对象由公共语言运行时 (CLR) 从 **throw** 语句点开始自动创建，因此必须从堆栈跟踪的开始点引发异常。

所有异常都包含一个名为 Message 的属性。应该设置此字符串来解释发生异常的原因。注意，不应将安全敏感信息放在消息文本中。除 Message 之外，ArgumentException 还包含一个名为 ParamName 的属性，应将该属性设置为导致引发异常的参数的名称。对于属性设置器，ParamName 应设置为 value。

公共的受保护方法应在其无法完成预期功能时引发异常。引发的异常类应该是符合错误条件的最确切的可用异常。这些异常应编写为类功能的一部分，派生类或对原始类的更新应保留相同的行为，以实现向后兼容性。

引发异常时要避免的情况

下表确定了在引发异常时要避免的做法：

- 不应使用异常来更改正常执行过程中的程序流程。异常只能用于报告和处理错误条件。
- 只能引发异常，而不能作为返回值或参数返回异常。
- 不要从自己的源代码中有意引发 System.Exception、System.SystemException、System.NullReferenceException 或 System.IndexOutOfRangeException。
- 不要创建可在调试模式下引发但不会在发布模式下引发的异常。若要在开发阶段确定运行时错误，请改用调试断言。

定义异常类

程序可以引发 System 命名空间中的预定义异常类（前面注明的情况除外），或通过从 Exception 派生来创建它们自己的异常类。派生类至少应定义四个构造函数：一个是默认构造函数，一个用来设置消息属性，一个用来设置 Message 属性和 InnerException 属性。第四个构造函数用于序列化异常。新异常类应该可序列化。例如：

```
[Serializable()]
public class InvalidDepartmentException : System.Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(messag
    public InvalidDepartmentException(string message, System.Except

    // A constructor is needed for serialization when an
    // exception propagates from a remoting server to the client.
    protected InvalidDepartmentException(System.Runtime.Serializati
        System.Runtime.Serialization.StreamingContext context) { }
}
```

仅当新属性提供的数据有助于解决异常时，才应将其添加到异常类。如果向派生的异常类添加了新属性，则应重写 ToString() 以返回添加的信息。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

异常和异常处理（C# 编程指南）

异常层次结构

异常处理（C# 编程指南）

编译器生成的异常（C# 编程指南）

有些异常在基本操作失败时由 .NET Framework 的公共语言运行时 (CLR) 自动引发。下表列出了这些异常和它们的错误条件。

异常	说明
ArithmeticException	在算术运算期间发生的异常（如 DivideByZeroException 和 OverflowException）的基类。
ArrayTypeMismatchException	当数组存储给定的元素时，如果由于该元素的实际类型与数组的实际类型不兼容而导致存储失败，就会引发此异常。
DivideByZeroException	在尝试用零除整数值时引发。
IndexOutOfRangeException	在尝试为数组设置小于零或超出数组界限的索引时引发。
InvalidCastException	当从基类型到接口或派生类型的显式转换在运行时失败时，就会引发此异常。
NullReferenceException	在尝试引用值为 null 的对象时引发。
OutOfMemoryException	在使用 new 运算符分配内存的尝试失败时引发。这表明可用于公共语言运行时的内存已耗尽。
OverflowException	在 checked 上下文中的算术运算溢出时引发。
StackOverflowException	当执行堆栈由于具有太多的挂起方法调用而耗尽时，就会引发此异常；这通常表明存在非常深的递归或无限递归。
TypeInitializationException	在静态构造函数引发异常并且不存在可以捕捉到它的兼容 catch 子句时引发。

请参阅

C# 编程指南

异常和异常处理（C# 编程指南）

异常处理（C# 编程指南）

try-catch（C# 参考）

try-finally（C# 参考）

try-catch-finally（C# 参考）

如何：使用 try/catch 处理异常（C# 编程指南）

try-catch 块的用途是捕捉和处理工作代码所生成的异常。 有些异常可以在 catch 块中处理，解决问题后不会再次引发异常；但更多情况下，您唯一能做的是确保引发适当的异常。

示例

在此示例中，IndexOutOfRangeException 不是最适当的异常：对本方法而言 ArgumentOutOfRangeException 更恰当些，因为错误是由调用方传入的 index 参数导致的。

```
class TestTryCatch
{
    static int GetInt(int[] array, int index)
    {
        try
        {
            return array[index];
        }
        catch (System.IndexOutOfRangeException e)  // CS0168
        {
            System.Console.WriteLine(e.Message);
            // Set IndexOutOfRangeException to the new exception's
            throw new System.ArgumentOutOfRangeException("index par
        }
    }
}
```

注释

导致异常的代码被括在 try 块中。 在其后面紧接着添加一个 catch 语句，以便在 IndexOutOfRangeException 发生时对其进行处理。 catch 块处理 IndexOutOfRangeException，并引发更适当的 ArgumentOutOfRangeException 异常。 为给调用方提供尽可能多的信息，应考虑将原始异常指定为新异常的 InnerException。 因为 InnerException 属性是只读，所以必须在新异常的构造函数中为其赋值。

请参见

参考

异常和异常处理（C# 编程指南） 异常处理（C# 编程指南）

概念

C# 编程指南

如何：使用 finally 执行清理代码（C# 编程指南）

finally 语句的目的是确保即使在引发异常的情况下也能立即进行必要的对象（通常是保存外部资源的对象）清理。此类清理功能的一个示例是在使用后立即对 FileStream 调用 Close，而不是等待公共语言运行时对该对象进行垃圾回收，如下所示：

```
static void CodeWithoutCleanup()
{
    System.IO.FileStream file = null;
    System.IO.FileInfo fileInfo = new System.IO.FileInfo("C:\\file

    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);

    file.Close();
}
```

为了将上面的代码转换为 **try-catch-finally** 语句，需要将清理代码与工作代码分开，如下所示。

```
static void CodeWithCleanup()
{
    System.IO.FileStream file = null;
    System.IO.FileInfo fileInfo = null;

    try
    {
        fileInfo = new System.IO.FileInfo("C:\\file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch(System.UnauthorizedAccessException e)
    {
        System.Console.WriteLine(e.Message);
    }
    finally
    {
        if (file != null)
        {
            file.Close();
        }
    }
}
```

因为在 OpenWrite() 调用前，**try** 块内随时都有可能发生异常，OpenWrite() 调用本身也有可能失败，所以我们无法保证该文件在我们尝试关闭它时处于打开状态。**finally** 块添加了一项检查，以确保在调用 Close 方法前，FileStream 对象不为 **null**。如果没有 **null** 检查，**finally** 块可能引发自身的 NullReferenceException，但是应当尽可能避免在 **finally** 块中引发异常。

在 **finally** 块中关闭数据库连接是另一个不错的选择。因为有时候数据库服务器允许的连接数是有限的，所以应尽快关闭数据库连接。在由于引发了异常而无法关闭连接的情况下，使用 **finally** 块也是比等待垃圾回收更好的一种选择。

请参阅

C# 编程指南

异常和异常处理（C# 编程指南）

异常处理（C# 编程指南）

using 语句（C# 参考）

try-catch（C# 参考）

try-finally（C# 参考）

try-catch-finally（C# 参考）

如何：捕捉非 CLS 异常

包括 C++/CLI 在内的某些 .NET 语言允许对象引发不是由 Exception 派生的异常。这类异常称为“非 CLS 异常”或“非异常”。在 Visual C# 中，您不能引发非 CLS 异常，但是可以采用两种方法捕捉这类异常：

- 在 catch (Exception e) 块中作为 RuntimeWrappedException 捕捉。

  默认情况下，Visual C# 程序集将非 CLS 异常作为包装异常来捕捉。如果需要访问可通过 WrappedException 属性访问的原始异常，请使用此方法。本主题后面的过程将解释如何使用此方式捕捉异常。

- 在位于 catch (Exception) 或 catch (Exception e) 块之后的常规 catch 块（没有指定异常类型的 catch 块）中。

  如果您要执行某个操作（如写入日志文件）以响应非 CLS 异常，并且不需要访问异常信息，请使用此方法。默认情况下，公共语言运行时会包装所有异常。要禁用此行为，请将程序集级别特性添加到您的代码中，该特性通常在 AssemblyInfo.cs 文件中。[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]

捕捉非 CLS 异常

1. 在 catch(Exception e) block 中，使用 **as** 关键字来测试 e 能否强制转换为 RuntimeWrappedException。
2. 通过 WrappedException 属性访问原始异常。

下面的示例演示如何捕捉由使用 C++/CLR 编写的类库引发的非 CLS 异常。请注意，在此示例中，Visual C# 客户端代码事先已经知道要引发的异常类型为 System.String。只要 WrappedException 属性的原始类型可通过您的代码来访问，就可以将该属性强制转换回其原始类型。

```
// Class library written in C++/CLR.
   ThrowNonCLS.Class1 myClass = new ThrowNonCLS.Class1();

   try
   {
    // throws gcnew System::String(
    // "I do not derive from System.Exception!");
    myClass.TestThrow();
   }

   catch (Exception e)
   {
    RuntimeWrappedException rwe = e as RuntimeWrappedException;
    if (rwe != null)
    {
      String s = rwe.WrappedException as String;
      if (s != null)
      {
        Console.WriteLine(s);
      }
    }
    else
    {
       // Handle other System.Exception types.
    }
   }
```

请参阅

RuntimeWrappedException

异常和异常处理（C# 编程指南）

文件系统和注册表（C# 编程指南）

以下主题说明如何使用 C# 和 .NET Framework 对文件、文件夹和注册表执行各种基本操作。

本节内容

标题	说明
如何：循环访问目录树（C# 编程指南）	说明如何手动循环访问目录树。
如何：获取有关文件、文件夹和驱动器的信息（C# 编程指南）	说明如何检索有关文件、文件夹和驱动器的信息，例如创建时间和大小。
如何：创建文件或文件夹（C# 编程指南）	说明如何创建新文件或文件夹。
如何：复制、删除和移动文件和文件夹（C# 编程指南）	说明如何复制、删除和移动文件和文件夹。
如何：提供文件操作进度对话框（C# 编程指南）	说明如何为某些文件操作显示标准的 Windows 进度对话框。
如何：写入文本文件（C# 编程指南）	说明如何写入文本文件。
如何：读取文本文件中的内容（C# 编程指南）	说明如何读取文本文件中的内容。
如何：一次一行地读取文本文件 (Visual C#)	说明如何一次从文件中检索一行文本。
如何：在注册表中创建注册表项 (Visual C#)	说明如何向系统注册表中写入项。

相关章节

文件和流 I/O

如何：复制、删除和移动文件和文件夹（C# 编程指南）

C# 编程指南

System.IO

如何：循环访问目录树（C# 编程指南）

词组“循环访问目录树”的意思是在指定的根文件夹下，访问每个嵌套子目录中任意深度的所有文件。您不必打开每一个文件。可以只检索 **string** 形式的文件名或子目录名，或者可以检索 System.IO.FileInfo 或 System.IO.DirectoryInfo 对象形式的其他信息。

> 注意
>
> 在 Windows 中，可交换使用术语“目录”和“文件夹”。大多数文档和用户界面文本使用术语“文件夹”，但 .NET Framework 类库使用术语“目录”。

最简单的例子，如果您确信您拥有指定根目录下所有目录的访问权限，则您可以使用 System.IO.SearchOption.AllDirectories 标志。该标志返回与指定模式相匹配的所有嵌套子目录。下面的示例演示如何使用该标志。

```
root.GetDirectories("*.*", System.IO.SearchOption.AllDirectories);
```

此方法的缺点是，如果指定根目录下任何一个子目录引发了 DirectoryNotFoundException 或 UnauthorizedAccessException，则整个方法将会失败并且不返回任何目录。使用 GetFiles 方法时也是如此。如果一定要处理特定子文件夹中的这些异常，则必须手动遍历该目录树，如下面的示例所示。

手动遍历目录树时，可以先处理子目录（前序遍历），或者可以先处理文件（后序遍历）。如果执行前序遍历，则在循环访问直接位于当前文件夹本身中的文件之前，请先遍历当前文件夹下的整个树。本文档后面的示例执行的是后序遍历，但您可以轻松地将它们修改为执行前序遍历。

另外还可以选择是使用递归遍历，还是使用基于堆栈的遍历。本文档后面的示例对这两种方法进行了演示。

如果必须对文件和文件夹上执行各种操作，则可以通过下面的方法将这些示例模块化：将操作重构到单独的函数，这样您就可以通过单个委托来调用这些函数。

> 注意
>
> NTFS 文件系统可以包含交接点、符号链接和硬链接形式的重新分析点。.NET Framework 方法（如 GetFiles 和 GetDirectories）将不返回重新分析点下的任何子目录。此行为可防止两个重新分析点相互引用时陷入无限循环。通常，为了确保您不会无意修改或删除文件，在使用重新分析点时应特别小心。如果需要精确控制重新分析点，请使用平台调用或本机代码直接调用相应的 Win32 文件系统方法。

下面的示例演示如何使用递归遍历目录树。递归方法很简洁，但如果目录树很大且嵌套很深，则有可能会引起堆栈溢出异常。

对于所处理的特定异常以及在每个文件和文件夹上执行的特定操作，都只是作为示例提供。您应该修改此代码来满足自己特定的需要。有关更多信息，请参见代码中的注释。

```
public class RecursiveFileSearch
{
    static System.Collections.Specialized.StringCollection log = ne

    static void Main()
    {
        // Start with drives if you have to search the entire compu
        string[] drives = System.Environment.GetLogicalDrives();

        foreach (string dr in drives)
        {
            System.IO.DriveInfo di = new System.IO.DriveInfo(dr);

            // Here we skip the drive if it is not ready to be read
            // is not necessarily the appropriate action in all sce
            if (!di.IsReady)
            {
                Console.WriteLine("The drive {0} could not be read"
                continue;
            }
            System.IO.DirectoryInfo rootDir = di.RootDirectory;
            WalkDirectoryTree(rootDir);
        }

        // Write out all the files that could not be processed.
        Console.WriteLine("Files with restricted access:");
        foreach (string s in log)
        {
            Console.WriteLine(s);
        }
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    static void WalkDirectoryTree(System.IO.DirectoryInfo root)
    {
        System.IO.FileInfo[] files = null;
        System.IO.DirectoryInfo[] subDirs = null;

        // First, process all the files directly under this folder
        try
        {
            files = root.GetFiles("*.*");
        }
        // This is thrown if even one of the files requires permiss
        // than the application provides.
        catch (UnauthorizedAccessException e)
```

```
            {
                // This code just writes out the message and continues
                // You may decide to do something different here. For e
                // can try to elevate your privileges and access the fi
                log.Add(e.Message);
            }

            catch (System.IO.DirectoryNotFoundException e)
            {
                Console.WriteLine(e.Message);
            }

            if (files != null)
            {
                foreach (System.IO.FileInfo fi in files)
                {
                    // In this example, we only access the existing Fil
                    // want to open, delete or modify the file, then
                    // a try-catch block is required here to handle the
                    // where the file has been deleted since the call t
                    Console.WriteLine(fi.FullName);
                }

                // Now find all the subdirectories under this directory
                subDirs = root.GetDirectories();

                foreach (System.IO.DirectoryInfo dirInfo in subDirs)
                {
                    // Resursive call for each subdirectory.
                    WalkDirectoryTree(dirInfo);
                }
            }
        }
    }
```

下面的示例演示在不使用递归的情况下如何循环访问目录树中的文件和文件夹。该技术使用泛型 Stack<T> 集合类型，该类型是一个后进先出 (LIFO) 堆栈。

对于所处理的特定异常以及在每个文件和文件夹上执行的特定操作，都只是作为示例提供。您应该修改此代码来满足自己特定的需要。有关更多信息，请参见代码中的注释。

```
public class StackBasedIteration
{
    static void Main(string[] args)
    {
        // Specify the starting folder on the command line, or in
        // Visual Studio in the Project > Properties > Debug pane.
        TraverseTree(args[0]);
```

```
            Console.WriteLine("Press any key");
            Console.ReadKey();
        }

        public static void TraverseTree(string root)
        {
            // Data structure to hold names of subfolders to be
            // examined for files.
            Stack<string> dirs = new Stack<string>(20);

            if (!System.IO.Directory.Exists(root))
            {
                throw new ArgumentException();
            }
            dirs.Push(root);

            while (dirs.Count > 0)
            {
                string currentDir = dirs.Pop();
                string[] subDirs;
                try
                {
                    subDirs = System.IO.Directory.GetDirectories(curren
                }
                // An UnauthorizedAccessException exception will be thr
                // discovery permission on a folder or file. It may or
                // to ignore the exception and continue enumerating the
                // folders. It is also possible (but unlikely) that a D
                // will be raised. This will happen if currentDir has b
                // another application or thread after our call to Dire
                // choice of which exceptions to catch depends entirely
                // you are intending to perform and also on how much yo
                // about the systems on which this code will run.
                catch (UnauthorizedAccessException e)
                {
                    Console.WriteLine(e.Message);
                    continue;
                }
                catch (System.IO.DirectoryNotFoundException e)
                {
                    Console.WriteLine(e.Message);
                    continue;
                }

                string[] files = null;
                try
                {
                    files = System.IO.Directory.GetFiles(currentDir);
                }

                catch (UnauthorizedAccessException e)
                {
```

```
                    Console.WriteLine(e.Message);
                    continue;
                }

                catch (System.IO.DirectoryNotFoundException e)
                {
                    Console.WriteLine(e.Message);
                    continue;
                }
                // Perform the required action on each file here.
                // Modify this block to perform your required task.
                foreach (string file in files)
                {
                    try
                    {
                        // Perform whatever action is required in your
                        System.IO.FileInfo fi = new System.IO.FileInfo(
                        Console.WriteLine("{0}: {1}, {2}", fi.Name, fi.
                    }
                    catch (System.IO.FileNotFoundException e)
                    {
                        // If file was deleted by a separate applicatio
                        //  or thread since the call to TraverseTree()
                        // then just continue.
                        Console.WriteLine(e.Message);
                        continue;
                    }
                }

                // Push the subdirectories onto the stack for traversal
                // This could also be done before handing the files.
                foreach (string str in subDirs)
                    dirs.Push(str);
            }
        }
    }
```

通常，测试每个文件夹以确定您的应用程序是否有权限打开它是非常耗时的。因此，代码示例只是将该部分操作放到了一个 **try/catch** 块内。您可以修改该 **catch** 块，以便在访问某个文件夹遭受拒绝时，您可以提升自己的权限，然后再次访问它。通常，只捕捉那些无需使应用程序停留在一个未知状态就可以处理的异常。

如果必须将目录树的内容存储到内存或磁盘中，则最好只存储每个文件的 FullName 属性（**string** 类型）。然后，可以根据需要使用该字符串创建一个新的 FileInfo 或 DirectoryInfo 对象，或者打开任何需要进行其他处理的文件。

可靠编程

可靠文件迭代代码必须考虑文件系统的诸多复杂性。有关更多信息，请参见 NTFS Technical Reference（NTFS 技术参考）。

请参阅

System.IO

LINQ and File Directories

文件系统和注册表（C# 编程指南）

如何：获取有关文件、文件夹和驱动器的信息（C# 编程指南）

在 .NET Framework 中，可以使用以下类来访问文件系统信息：

- System.IO.FileInfo
- System.IO.DirectoryInfo
- System.IO.DriveInfo
- System.IO.Directory
- System.IO.File

FileInfo 和 DirectoryInfo 类表示文件或目录，包含公开 NTFS 文件系统所支持的很多文件特性的属性，同时还包含用于打开、关闭、移动和删除文件和文件夹的方法。可以通过将表示文件、文件夹或驱动器名称的字符串传递到下面的构造函数来创建这些类的实例：

```
System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
```

此外，还可以通过调用 DirectoryInfo.GetDirectories、DirectoryInfo.GetFiles 和 DriveInfo.RootDirectory 来获取文件、文件夹或驱动器的名称。

System.IO.Directory 和 System.IO.File 类提供用于检索有关目录和文件的信息的静态方法。

下面的示例演示访问有关文件和文件夹的信息的各种方法。

```
class FileSysInfo
{
    static void Main()
    {
        // You can also use System.Environment.GetLogicalDrives to
        // obtain names of all logical drives on the computer.
        System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
        Console.WriteLine(di.TotalFreeSpace);
        Console.WriteLine(di.VolumeLabel);

        // Get the root directory and print out some information ab
        System.IO.DirectoryInfo dirInfo = di.RootDirectory;
        Console.WriteLine(dirInfo.Attributes.ToString());

        // Get the files in the directory and print out some inforr
        System.IO.FileInfo[] fileNames = dirInfo.GetFiles("*.*");
```

```
        foreach (System.IO.FileInfo fi in fileNames)
        {
            Console.WriteLine("{0}: {1}: {2}", fi.Name, fi.LastAcce
        }

        // Get the subdirectories directly that is under the root.
        // See "How to: Iterate Through a Directory Tree" for an ex
        // iterate through an entire tree.
        System.IO.DirectoryInfo[] dirInfos = dirInfo.GetDirectories

        foreach (System.IO.DirectoryInfo d in dirInfos)
        {
            Console.WriteLine(d.Name);
        }

        // The Directory and File classes provide several static me
        // for accessing files and directories.

        // Get the current application directory.
        string currentDirName = System.IO.Directory.GetCurrentDirec
        Console.WriteLine(currentDirName);

        // Get an array of file names as strings rather than FileIn
        // Use this method when storage space is an issue, and when
        // hold on to the file name reference for a while before yo
        // the file.
        string[] files = System.IO.Directory.GetFiles(currentDirNam

        foreach (string s in files)
        {
            // Create the FileInfo object only when needed to ensur
            // the information is as current as possible.
            System.IO.FileInfo fi = null;
            try
            {
                 fi = new System.IO.FileInfo(s);
            }
            catch (System.IO.FileNotFoundException e)
            {
                // To inform the user and continue is
                // sufficient for this demonstration.
                // Your application may require different behavior.
                Console.WriteLine(e.Message);
                continue;
            }
            Console.WriteLine("{0} : {1}",fi.Name, fi.Directory);
        }

        // Change the directory. In this case, first check to see
        // whether it already exists, and create it if it does not.
        // If this is not appropriate for your application, you can
        // handle the System.IO.IOException that will be raised if
        // directory cannot be found.
```

```
            if (!System.IO.Directory.Exists(@"C:\Users\Public\TestFolde
            {
                System.IO.Directory.CreateDirectory(@"C:\Users\Public\T
            }

            System.IO.Directory.SetCurrentDirectory(@"C:\Users\Public\T

            currentDirName = System.IO.Directory.GetCurrentDirectory();
            Console.WriteLine(currentDirName);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
```

可靠编程

处理用户指定的路径字符串时，还应处理在以下情况下引发的异常：

- 文件名的格式不正确。例如，包含无效字符或仅包含空白。
- 文件名为空。
- 文件名长于系统定义的最大长度。
- 文件名包含冒号 (:)。

如果应用程序不具有读取指定文件所需的足够权限，则无论路径是否存在，**Exists** 方法都将返回 **false**；该方法不引发异常。

请参阅

System.IO

C# 编程指南

文件系统和注册表（C# 编程指南）

如何：创建文件或文件夹（C# 编程指南）

您可通过编程方式在您的计算机上创建文件夹、子文件夹和子文件夹中的文件，并将数据写入文件。

```
public class CreateFileOrFolder
{
    static void Main()
    {
        // Specify a name for your top-level folder.
        string folderName = @"c:\Top-Level Folder";

        // To create a string that specifies the path to a subfolde
        // top-level folder, add a name for the subfolder to folder
        string pathString = System.IO.Path.Combine(folderName, "Sub

        // You can write out the path name directly instead of usin
        // method. Combine just makes the process easier.
        string pathString2 = @"c:\Top-Level Folder\SubFolder2";

        // You can extend the depth of your path if you want to.
        //pathString = System.IO.Path.Combine(pathString, "SubSubFo

        // Create the subfolder. You can verify in File Explorer th
        // structure in the C: drive.
        //    Local Disk (C:)
        //        Top-Level Folder
        //            SubFolder
        System.IO.Directory.CreateDirectory(pathString);

        // Create a file name for the file you want to create.
        string fileName = System.IO.Path.GetRandomFileName();

        // This example uses a random string for the name, but you
        // a particular name.
        //string fileName = "MyNewFile.txt";

        // Use Combine again to add the file name to the path.
        pathString = System.IO.Path.Combine(pathString, fileName);

        // Verify the path that you have constructed.
        Console.WriteLine("Path to my file: {0}\n", pathString);

        // Check that the file doesn't already exist. If it doesn't
        // the file and write integers 0 - 99 to it.
        // DANGER: System.IO.File.Create will overwrite the file if
        // This could happen even with random file names, although
        if (!System.IO.File.Exists(pathString))
        {
```

```
                using (System.IO.FileStream fs = System.IO.File.Create(
                {
                    for (byte i = 0; i < 100; i++)
                    {
                        fs.WriteByte(i);
                    }
                }
            }
            else
            {
                Console.WriteLine("File \"{0}\" already exists.", fileN
                return;
            }

            // Read and display the data from your file.
            try
            {
                byte[] readBuffer = System.IO.File.ReadAllBytes(pathStr
                foreach (byte b in readBuffer)
                {
                    Console.Write(b + " ");
                }
                Console.WriteLine();
            }
            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
            }

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
        // Sample output:

        // Path to my file: c:\Top-Level Folder\SubFolder\ttxvauxe.vv0

        //0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
        //30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 5
        // 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76
        //3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
    }
```

如果该文件夹已存在，则 CreateDirectory 不执行任何操作，且不会引发异常。但是，File.Create 用新的文件替换现有文件。该示例使用一个 **if-else** 语句阻止现有文件被替换。

通过在示例中做出以下更改，您可以根据具有某个名称的程序是否存在来指定不同的结果。如果该文件不存在，代码将创建一个文件。如果该文件存在，代码将把数据添加到该文件中。

- 指定一个非随机文件名。

```
// Comment out the following line.
//string fileName = System.IO.Path.GetRandomFileName();

// Replace that line with the following assignment.
string fileName = "MyNewFile.txt";
```

- 用以下代码中的 **using** 语句替换 **if-else** 语句。

```
using (System.IO.FileStream fs = new System.IO.FileStream(pathS
{
    for (byte i = 0; i &lt; 100; i++)
    {
        fs.WriteByte(i);
    }
}
```

运行该示例若干次以验证数据是否每次都添加到文件中。

有关更多可以尝试的 **FileMode** 值的信息，请参阅 FileMode。

以下情况可能会导致异常：

- 文件夹名称格式不正确。例如，它包含非法字符或仅仅是空白（ArgumentException 类）。使用 Path 类创建有效路径名。
- 要创建的文件夹的父文件夹是只读的（IOException 类）。
- 文件夹名称是 **null**（ArgumentNullException 类）。
- 文件夹名称太长（PathTooLongException 类）。
- 文件夹名称只是冒号“:”（PathTooLongException 类）。

.NET Framework 安全性

在部分信任的情况下可能会引发 SecurityException 类的实例。

如果没有创建文件夹的权限，则该示例引发 UnauthorizedAccessException 类的实例。

请参阅

System.IO

C# 编程指南

文件系统和注册表（C# 编程指南）

如何：复制、删除和移动文件和文件夹（C# 编程指南）

以下示例说明如何使用 System.IO 命名空间中的 System.IO.File、System.IO.Directory、System.IO.FileInfo 和 System.IO.DirectoryInfo 类以同步方式复制、移动和删除文件和文件夹。 这些示例没有提供进度栏或其他任何用户界面。 如果您想提供一个标准进度对话框，请参见如何：提供文件操作进度对话框（C# 编程指南）。

在操作多个文件时，请使用 System.IO.FileSystemWatcher 提供一些事件，以便可以利用这些事件计算进度。 另一种方法是使用平台调用来调用 Windows Shell 中相应的文件相关方法。 有关如何异步执行这些文件操作的信息，请参见异步文件 I/O。

示例

下面的示例演示如何复制文件和目录。

```
// Simple synchronous file copy operations with no user interface.
// To run this sample, first create the following directories and f
// C:\Users\Public\TestFolder
// C:\Users\Public\TestFolder\test.txt
// C:\Users\Public\TestFolder\SubDir\test.txt
public class SimpleFileCopy
{
    static void Main()
    {
        string fileName = "test.txt";
        string sourcePath = @"C:\Users\Public\TestFolder";
        string targetPath =  @"C:\Users\Public\TestFolder\SubDir";

        // Use Path class to manipulate file and directory paths.
        string sourceFile = System.IO.Path.Combine(sourcePath, file
        string destFile = System.IO.Path.Combine(targetPath, fileNa

        // To copy a folder's contents to a new location:
        // Create a new target folder, if necessary.
        if (!System.IO.Directory.Exists(targetPath))
        {
            System.IO.Directory.CreateDirectory(targetPath);
        }

        // To copy a file to another location and
        // overwrite the destination file if it already exists.
        System.IO.File.Copy(sourceFile, destFile, true);
```

```
        // To copy all the files in one directory to another direct
        // Get the files in the source folder. (To recursively iter
        // all subfolders under the current directory, see
        // "How to: Iterate Through a Directory Tree.")
        // Note: Check for target path was performed previously
        //       in this code example.
        if (System.IO.Directory.Exists(sourcePath))
        {
            string[] files = System.IO.Directory.GetFiles(sourcePat

            // Copy the files and overwrite destination files if th
            foreach (string s in files)
            {
                // Use static Path methods to extract only the file
                fileName = System.IO.Path.GetFileName(s);
                destFile = System.IO.Path.Combine(targetPath, fileN
                System.IO.File.Copy(s, destFile, true);
            }
        }
        else
        {
            Console.WriteLine("Source path does not exist!");
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

下面的示例演示如何移动文件和目录。

```
// Simple synchronous file move operations with no user interface.
public class SimpleFileMove
{
    static void Main()
    {
        string sourceFile = @"C:\Users\Public\public\test.txt";
        string destinationFile = @"C:\Users\Public\private\test.txt

        // To move a file or folder to a new location:
        System.IO.File.Move(sourceFile, destinationFile);

        // To move an entire directory. To programmatically modify
        // path strings, use the System.IO.Path class.
        System.IO.Directory.Move(@"C:\Users\Public\public\test\", @
    }
}
```

下面的示例演示如何删除文件和目录。

```
// Simple synchronous file deletion operations with no user interfa
// To run this sample, create the following files on your drive:
// C:\Users\Public\DeleteTest\test1.txt
// C:\Users\Public\DeleteTest\test2.txt
// C:\Users\Public\DeleteTest\SubDir\test2.txt

public class SimpleFileDelete
{
    static void Main()
    {
        // Delete a file by using File class static method...
        if(System.IO.File.Exists(@"C:\Users\Public\DeleteTest\test.
        {
            // Use a try block to catch IOExceptions, to
            // handle the case of the file already being
            // opened by another process.
            try
            {
                System.IO.File.Delete(@"C:\Users\Public\DeleteTest\
            }
            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
                return;
            }
        }

        // ...or by using FileInfo instance method.
        System.IO.FileInfo fi = new System.IO.FileInfo(@"C:\Users\P
```

```
        try
        {
            fi.Delete();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Delete a directory. Must be writable or empty.
        try
        {
            System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }
        // Delete a directory and all subdirectories with Directory
        if(System.IO.Directory.Exists(@"C:\Users\Public\DeleteTest'
        {
            try
            {
                System.IO.Directory.Delete(@"C:\Users\Public\Delete
            }

            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
            }
        }

        // ...or with DirectoryInfo instance method.
        System.IO.DirectoryInfo di = new System.IO.DirectoryInfo(@'
        // Delete this dir and all subdirs.
        try
        {
            di.Delete(true);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

    }
}
```

请参见

任务

如何：提供文件操作进度对话框（C# 编程指南）

参考

System.IO

概念

C# 编程指南 通用 I/O 任务

其他资源

文件系统和注册表（C# 编程指南） 文件和流 I/O

如何：提供文件操作进度对话框（C# 编程指南）

如果您使用 Microsoft.VisualBasic 命名空间里的 CopyFile(String, String, UIOption) 方法，则可以提供能够显示 Windows 中文件操作进程的标准对话框。

> 注意
>
> 以下说明中的某些 Visual Studio 用户界面元素在计算机上出现的名称或位置可能会不同。 这些元素取决于你所使用的 Visual Studio 版本和你所使用的设置。 有关详细信息，请参阅个性化 Visual Studio IDE。

在 Visual Studio 中添加引用

1. 在菜单栏上，依次选择“项目”、“添加引用”。

   出现“引用管理器”对话框。

2. 在“程序集”区域中，选择“框架”（如果尚未选择）。

3. 在名称列表中，选中“Microsoft.VisualBasic”复选框，然后选择“确定”按钮关闭对话框。

下面的代码将 sourcePath 指定的目录复制到 destinationPath 指定的目录中。此代码还提供一个标准对话框，该对话框显示预计完成操作还需要的时间量。

```
// The following using directive requires a project reference to Mi
using Microsoft.VisualBasic.FileIO;

class FileProgress
{
    static void Main()
    {
        // Specify the path to a folder that you want to copy. If t
        // you won't have time to see the progress dialog box.
        string sourcePath = @"C:\Windows\symbols\";
        // Choose a destination for the copied files.
        string destinationPath = @"C:\TestFolder";

        FileSystem.CopyDirectory(sourcePath, destinationPath,
            UIOption.AllDialogs);
    }
}
```

请参阅

文件系统和注册表（C# 编程指南）

如何：写入文本文件（C# 编程指南）

以下示例给出了将文本写入文件的各种方法。 前两个示例对 System.IO.File 类使用静态便捷方法以将任何 IEnumerable<string> 的每个元素和一个字符串写入文本文件。 示例 3 展示了在写入文件时必须分别处理文本的每一行时，如何将文本添加到文件。 示例 1-3 覆盖文件中的所有现有内容，但示例 4 展示如何将文本追加到现有文件。

这些示例均会将字符串文本写入文件，但是你更有可能需要使用 Format 方法，此方法具有很多用于写入不同类型值的控件，包括在字段中左右对齐、有无边距等。还可以使用 C# 字符串内插功能。

示例

```
class WriteTextFile
{
    static void Main()
    {

        // These examples assume a "C:\Users\Public\TestFolder" fol
        // You can modify the path if necessary.

        // Example #1: Write an array of strings to a file.
        // Create a string array that consists of three lines.
        string[] lines = { "First line", "Second line", "Third line
        // WriteAllLines creates a file, writes a collection of str
        // and then closes the file.  You do NOT need to call Flush
        System.IO.File.WriteAllLines(@"C:\Users\Public\TestFolder\W

        // Example #2: Write one string to a text file.
        string text = "A class is the most powerful data type in C#
                      "a class defines the data and behavior of th
        // WriteAllText creates a file, writes the specified string
        // and then closes the file.    You do NOT need to call Flu
        System.IO.File.WriteAllText(@"C:\Users\Public\TestFolder\Wr

        // Example #3: Write only some strings in an array to a fil
        // The using statement automatically flushes AND CLOSES the
        // IDisposable.Dispose on the stream object.
        // NOTE: do not use FileStream for text files because it wr
        // encodes the output as text.
        using (System.IO.StreamWriter file =
            new System.IO.StreamWriter(@"C:\Users\Public\TestFolder
        {
            foreach (string line in lines)
            {
```

```
                    // If the line doesn't contain the word 'Second', w
                    if (!line.Contains("Second"))
                    {
                        file.WriteLine(line);
                    }
                }
            }

            // Example #4: Append new text to an existing file.
            // The using statement automatically flushes AND CLOSES the
            // IDisposable.Dispose on the stream object.
            using (System.IO.StreamWriter file =
                new System.IO.StreamWriter(@"C:\Users\Public\TestFolder
            {
                file.WriteLine("Fourth line");
            }
        }
    }
 //Output (to WriteLines.txt):
 //   First line
 //   Second line
 //   Third line

 //Output (to WriteText.txt):
 //   A class is the most powerful data type in C#. Like a structur

 //Output to WriteLines2.txt after Example #3:
 //   First line
 //   Third line

 //Output to WriteLines2.txt after Example #4:
 //   First line
 //   Third line
 //   Fourth line
```

这些示例均会将字符串文本写入文件，但是你更有可能需要使用 Format 方法，此方法具有很多用于写入不同类型值的控件，包括在字段中左右对齐、有无边距等。还可以使用 C# 字符串内插功能。

可靠编程

以下情况可能会导致异常：

- 文件已存在并且为只读。
- 路径名可能太长。
- 磁盘可能已满。

请参见

概念

C# 编程指南

其他资源

文件系统和注册表（C# 编程指南） 示例：将集合保存到应用程序存储

如何：读取文本文件中的内容（C# 编程指南）

此示例读取文本文件的内容以使用 System.IO.File 选件类的静态方法 ReadAllText 和 ReadAllLines。

有关使用 StreamReader 的示例，请参见如何：一次一行地读取文本文件 (Visual C#)。

注意
本示例的文件在主题 如何：写入文本文件（C# 编程指南） 创建。

```
class ReadFromFile
{
    static void Main()
    {
        // The files used in this example are created in the topic
        // How to: Write to a Text File. You can change the path an
        // file name to substitute text files of your own.

        // Example #1
        // Read the file as one string.
        string text = System.IO.File.ReadAllText(@"C:\Users\Public\

        // Display the file contents to the console. Variable text
        System.Console.WriteLine("Contents of WriteText.txt = {0}",

        // Example #2
        // Read each line of the file into a string array. Each ele
        // of the array is one line of the file.
        string[] lines = System.IO.File.ReadAllLines(@"C:\Users\Pub

        // Display the file contents by using a foreach loop.
        System.Console.WriteLine("Contents of WriteLines2.txt = ");
        foreach (string line in lines)
        {
            // Use a tab to indent each line of the file.
            Console.WriteLine("\t" + line);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

编译代码

将代码复制并粘贴到 C# 控制台应用程序。

如果不使用从 如何：写入文本文件（C# 编程指南）的文本文件，请替换为您计算机上的参数。**ReadAllText** 和 **ReadAllLines** 使用适当的路径和文件名。

可靠编程

以下情况可能会导致异常：

- 文件不在指定的位置存在或不存在。检查路径和文件名的拼写。

.NET Framework 安全性

不要依赖文件名来确定文件的内容。例如，文件 myFile.cs 可能不是 C# 源文件。

请参阅

System.IO

C# 编程指南

文件系统和注册表（C# 编程指南）

如何：一次一行地读取文本文件 (Visual C#)

本示例使用 **StreamReader** 类的 **ReadLine** 方法将文本文件的内容读取（一次读取一行）到字符串中。所有文本行都保存在字符串 line 中并显示在屏幕上。

```
int counter = 0;
string line;

// Read the file and display it line by line.
System.IO.StreamReader file =
    new System.IO.StreamReader(@"c:\test.txt");
while((line = file.ReadLine()) != null)
{
    System.Console.WriteLine (line);
    counter++;
}

file.Close();
System.Console.WriteLine("There were {0} lines.", counter);
// Suspend the screen.
System.Console.ReadLine();
```

编译代码

复制该代码，并将其粘贴到某控制台应用程序的 **Main** 方法中。

将 "c:\test.txt" 替换为实际的文件名。

可靠编程

以下情况可能会导致异常：

- 该文件可能不存在。

.NET Framework 安全性

不要根据文件的名称来判断文件的内容。例如，文件 myFile.cs 可能不是 C# 源文件。

请参阅

System.IO

C# 编程指南

文件系统和注册表（C# 编程指南）

如何：在注册表中创建注册表项 (Visual C#)

本示例将值对“Name”和“Isabella”添加到当前用户的注册表中的注册表项“Names”之下。

```
Microsoft.Win32.RegistryKey key;
key = Microsoft.Win32.Registry.CurrentUser.CreateSubKey("Names");
key.SetValue("Name", "Isabella");
key.Close();
```

编译代码

- 复制该代码，并将其粘贴到某控制台应用程序的 **Main** 方法中。
- 将 *Names* 参数替换为直接存在于注册表 HKEY_CURRENT_USER 节点下的项的名称。
- 将 _Nam_e 参数替换为直接存在于“Names”节点下的值的名称。

可靠编程

检查注册表结构以查找适合您的项的位置。例如，您可能需要打开当前用户的 Software 项，并且用您的公司的名称创建一项。然后将注册表值添加到您的公司的项上。

以下情况可能会导致异常：

- 注册表项的名称为空。
- 用户没有创建注册表项的权限。
- 注册表项名称超过 255 个字符的限制。
- 注册表项已关闭。
- 注册表项是只读的。

.NET Framework 安全性

与将数据写入本地计算机 (Microsoft.Win32.Registry.LocalMachine) 相比，将数据写入用户文件夹 (Microsoft.Win32.Registry.CurrentUser) 更安全。

在您创建注册表值时，需要确定如果该值已存在则应执行的操作。另一进程（可能是恶意进程）可能已创建了该值，并拥有对该值的访问权。将数据放入注册表值后，其他进程就可以使用这些数据了。若要防止出现这种情况，请使用 **Overload:Microsoft.Win32.RegistryKey.GetValue** 方法。如果注册表项不存在，该方法将返回 null。

即使注册表项受访问控制列表 (ACL) 的保护，在注册表中以纯文本形式存储机密信息（例如密码）也不安全。

请参阅

System.IO

C# 编程指南

文件系统和注册表（C# 编程指南）

Read, write and delete from the registry with C#

泛型（C# 编程指南）

2.0 版 C# 语言和公共语言运行时 (CLR) 中增加了泛型。泛型将类型参数的概念引入 .NET Framework，类型参数使得设计如下类和方法成为可能：这些类和方法将一个或多个类型的指定推迟到客户端代码声明并实例化该类或方法的时候。例如，通过使用泛型类型参数 T，您可以编写其他客户端代码能够使用的单个类，而不致引入运行时强制转换或装箱操作的成本或风险，如下所示：

```
// Declare the generic class.
public class GenericList<T>
{
    void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleCl
    }
}
```

泛型概述

- 使用泛型类型可以最大限度地重用代码、保护类型的安全以及提高性能。
- 泛型最常见的用途是创建集合类。
- .NET Framework 类库在 System.Collections.Generic 命名空间中包含几个新的泛型集合类。应尽可能地使用这些类来代替普通的类，如 System.Collections 命名空间中的 ArrayList。
- 您可以创建自己的泛型接口、泛型类、泛型方法、泛型事件和泛型委托。
- 可以对泛型类进行约束以访问特定数据类型的方法。
- 关于泛型数据类型中使用的类型的信息可在运行时通过使用反射获取。

相关章节

有关更多信息：

- 泛型介绍（C# 编程指南）
- 泛型的优点（C# 编程指南）
- 泛型类型参数（C# 编程指南）
- 类型参数的约束（C# 编程指南）
- 泛型类（C# 编程指南）
- 泛型接口（C# 编程指南）
- 泛型方法（C# 编程指南）
- 泛型委托（C# 编程指南）
- 泛型代码中的默认关键字（C# 编程指南）
- C++ 模板和 C# 泛型之间的区别（C# 编程指南）
- 泛型和反射（C# 编程指南）
- 运行时中的泛型（C# 编程指南）
- .NET Framework 类库中的泛型（C# 编程指南）

C# 语言规范

有关更多信息，请参见 C# 语言规范。

请参阅

System.Collections.Generic

C# 编程指南

类型（C# 编程指南）

<typeparam>（C# 编程指南）

<typeparamref>（C# 编程指南）

泛型介绍（C# 编程指南）

泛型类和泛型方法同时具备可重用性、类型安全和效率，这是非泛型类和非泛型方法无法具备的。泛型通常用与集合以及作用于集合的方法一起使用。.NET Framework 2.0 版类库提供一个新的命名空间 System.Collections.Generic，其中包含几个新的基于泛型的集合类。建议面向 .NET Framework 2.0 及更高版本的所有应用程序都使用新的泛型集合类，而不要使用旧的非泛型集合类如 ArrayList。有关更多信息，请参见 .NET Framework 类库中的泛型（C# 编程指南）。

当然，也可以创建自定义泛型类型和方法，以提供自己的通用解决方案，设计类型安全的高效模式。下面的代码示例演示一个用于演示用途的简单泛型链接列表类。（大多数情况下，应使用 .NET Framework 类库提供的 List<T> 类，而不是自行创建类。）在通常使用具体类型来指示列表中存储的项的类型的场合，可使用类型参数 *T*。其使用方法如下：

- 在 **AddHead** 方法中作为方法参数的类型。
- 在 **Node** 嵌套类中作为公共方法 **GetNext** 和 **Data** 属性的返回类型。
- 在嵌套类中作为私有成员数据的类型。

注意，T 可用于 **Node** 嵌套类。如果使用具体类型实例化 GenericList<T>（例如，作为 GenericList<int>），则所有的 T 都将被替换为 *int*。

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
```

```
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

下面的代码示例演示客户端代码如何使用泛型 GenericList<T> 类来创建整数列表。只需更改类型参数，即可方便地修改下面的代码示例，创建字符串或任何其他自定义类型的列表：

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

请参阅

System.Collections.Generic

C# 编程指南

泛型（C# 编程指南）

泛型的优点（C# 编程指南）

在公共语言运行时和 C# 语言的早期版本中，通用化是通过在类型与通用基类型 Object 之间进行强制转换来实现的，泛型提供了针对这种限制的解决方案。通过创建泛型类，您可以创建一个在编译时类型安全的集合。

使用非泛型集合类的限制可以通过编写一小段程序来演示，该程序使用 .NET Framework 类库中的 ArrayList 集合类。 ArrayList 是一个使用起来非常方便的集合类，无需进行修改即可用来存储任何引用或值类型。

```
// The .NET Framework 1.1 way to create a list:
System.Collections.ArrayList list1 = new System.Collections.ArrayLi
list1.Add(3);
list1.Add(105);

System.Collections.ArrayList list2 = new System.Collections.ArrayLi
list2.Add("It is raining in Redmond.");
list2.Add("It is snowing in the mountains.");
```

但这种方便是需要付出代价的。添加到 ArrayList 中的任何引用或值类型都将隐式地向上强制转换为 Object。如果项是值类型，则必须在将其添加到列表中时进行装箱操作，在检索时进行取消装箱操作。强制转换以及装箱和取消装箱操作都会降低性能；在必须对大型集合进行循环访问的情况下，装箱和取消装箱的影响非常明显。

另一个限制是缺少编译时类型检查；因为 ArrayList 会将所有项都强制转换为 Object，所以在编译时无法防止客户端代码执行类似如下的操作：

```
System.Collections.ArrayList list = new System.Collections.ArrayLis
// Add an integer to the list.
list.Add(3);
// Add a string to the list. This will compile, but may cause an e
list.Add("It is raining in Redmond.");

int t = 0;
// This causes an InvalidCastException to be returned.
foreach (int x in list)
{
    t += x;
}
```

尽管将字符串和 **ints** 组合在一个 ArrayList 中的做法在创建异类集合时是完全可接受的，并且有时需要有意为之，但这种做法很可能产生编程错误，并且直到运行时才能检测到此错误。

在 C# 语言的 1.0 和 1.1 版本中，只能通过编写自己的特定于类型的集合来避免 .NET Framework 基类库集合类中的通用代码的危险。当然，由于此类不可对多个数据类型重用，因此将丧失通用化的优点，并且您必须对要存储的每个类型重新编写该类。

ArrayList 和其他相似类真正需要的是：客户端代码基于每个实例指定这些类要使用的具体数据类型的方式。这样将不再需要向上强制转换为 **T:System.Object**，同时，也使得编译器可以进行类型检查。换句话说，ArrayList 需要一个类型参数。这正是泛型所能提供的。在 **N:System.Collections.Generic** 命名空间的泛型 List<T> 集合中，向集合添加项的操作类似于以下形式：

```
// The .NET Framework 2.0 way to create a list
List<int> list1 = new List<int>();

// No boxing, no casting:
list1.Add(3);

// Compile-time error:
// list1.Add("It is raining in Redmond.");
```

对于客户端代码，与 ArrayList 相比，使用 List<T> 时添加的唯一语法是声明和实例化中的类型参数。虽然这种方式稍微增加了编码的复杂性，但好处是您可以创建一个比 ArrayList 更安全并且速度更快的列表，对于列表项是值类型的情况尤为如此。

请参阅

System.Collections.Generic

C# 编程指南

泛型介绍（C# 编程指南）

装箱和取消装箱（C# 编程指南）

Collections Best Practices（有关集合的最佳做法）

泛型类型参数（C# 编程指南）

在泛型类型或方法定义中，类型参数是客户端在实例化泛型类型的变量时指定的特定类型的占位符。泛型类（如 泛型介绍（C# 编程指南） 中列出的 GenericList<T>）不可以像这样使用，因为它实际上并不是一个类型，而更像是一个类型的蓝图。若要使用 GenericList<T>，客户端代码必须通过指定尖括号中的类型参数来声明和实例化构造类型。此特定类的类型参数可以是编译器识别的任何类型。可以创建任意数目的构造类型实例，每个实例使用不同的类型参数，如下所示：

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>(
```

在每个 GenericList<T> 实例中，类中出现的每个 T 都会在运行时替换为相应的类型参数。通过这种替换方式，我们使用一个类定义创建了三个独立的类型安全的有效对象。有关 CLR 如何执行此替换的更多信息，请参见运行时中的泛型（C# 编程指南）。

类型参数命名准则

- 务必使用描述性名称命名泛型类型参数，除非单个字母名称完全可以让人了解它表示的含义，而描述性名称不会有更多的意义。

```
public interface ISessionChannel&lt;TSession&gt; { /*...*/ }
public delegate TOutput Converter&lt;TInput, TOutput&gt;(TInput
public class List&lt;T&gt; { /*...*/ }
```

- 考虑使用 T 作为具有单个字母类型参数的类型的类型参数名。

```
public int IComparer&lt;T&gt;() { return 0; }
public delegate bool Predicate&lt;T&gt;(T item);
public struct Nullable&lt;T&gt; where T : struct { /*...*/ }
```

- 务必将“T”作为描述性类型参数名的前缀。

```
public interface ISessionChannel&lt;TSession&gt;
{
    TSession Session { get; }
}
```

- 考虑在参数名中指示对此类型参数的约束。例如，可以将带有 ISession 约束的参数命名为 TSession。

请参阅

System.Collections.Generic

C# 编程指南

泛型（C# 编程指南）

C++ 模板和 C# 泛型之间的区别（C# 编程指南）

类型参数的约束（C# 编程指南）

在定义泛型类时，可以对客户端代码能够在实例化类时用于类型参数的类型种类施加限制。如果客户端代码尝试使用某个约束所不允许的类型来实例化类，则会产生编译时错误。这些限制称为约束。约束是使用 **where** 上下文关键字指定的。下表列出了六种类型的约束：

约束	说明
T：结构	类型参数必须是值类型。可以指定除 Nullable 以外的任何值类型。有关更多信息，请参见使用可以为 null 的类型（C# 编程指南）。
T：类	类型参数必须是引用类型；这一点也适用于任何类、接口、委托或数组类型。
T：new()	类型参数必须具有无参数的公共构造函数。当与其他约束一起使用时，**new()** 约束必须最后指定。
T：<基类名>	类型参数必须是指定的基类或派生自指定的基类。
T：<接口名称>	类型参数必须是指定的接口或实现指定的接口。可以指定多个接口约束。约束接口也可以是泛型的。
T：U	为 T 提供的类型参数必须是为 U 提供的参数或派生自为 U 提供的参数。

使用约束的原因

如果要检查泛型列表中的某个项以确定它是否有效，或者将它与其他某个项进行比较，则编译器必须在一定程度上保证它需要调用的运算符或方法将受到客户端代码可能指定的任何类型参数的支持。这种保证是通过对泛型类定义应用一个或多个约束获得的。例如，基类约束告诉编译器：仅此类型的对象或从此类型派生的对象才可用作类型参数。一旦编译器有了这个保证，它就能够允许在泛型类中调用该类型的方法。约束是使用上下文关键字 **where** 应用的。下面的代码示例演示可通过应用基类约束添加到 GenericList<T> 类（在泛型介绍（C# 编程指南）中）的功能。

```
public class Employee
{
    private string name;
    private int id;

    public Employee(string s, int i)
    {
        name = s;
```

```
        id = i;
    }

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public int ID
    {
        get { return id; }
        set { id = value; }
    }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        private Node next;
        private T data;

        public Node(T t)
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    public GenericList() //constructor
    {
        head = null;
    }

    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
```

```
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    public T FindFirstOccurrence(string s)
    {
        Node current = head;
        T t = null;

        while (current != null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                t = current.Data;
                break;
            }
            else
            {
                current = current.Next;
            }
        }
        return t;
    }
}
```

约束使得泛型类能够使用 Employee.Name 属性，因为类型为 T 的所有项都保证是 Employee 对象或从 Employee 继承的对象。

可以对同一类型参数应用多个约束，并且约束自身可以是泛型类型，如下所示：

```
class EmployeeList<T> where T : Employee, IEmployee, System.ICompar
{
    // ...
}
```

通过约束类型参数，可以增加约束类型及其继承层次结构中的所有类型所支持的允许操作和方法调用的数量。因此，在设计泛型类或方法时，如果要对泛型成员执行除简单赋值之外的任何操作或调用 **System.Object** 不支持的任何方法，您将需要对该类型参数应用约束。

在应用 where T : class 约束时，避免对类型参数使用 **==** 和 **!=** 运算符，因为这些运算符仅测试引用同一性而不测试值相等性。即使在用作参数的类型中重载这些运算符也是如此。下面的代码说明了这一点；即使 String 类重载 **==** 运算符，输出也为 false。

```
public static void OpTest<T>(T s, T t) where T : class
{
    System.Console.WriteLine(s == t);
}
static void Main()
{
    string s1 = "target";
    System.Text.StringBuilder sb = new System.Text.StringBuilder("t
    string s2 = sb.ToString();
    OpTest<string>(s1, s2);
}
```

这种情况的原因在于，编译器在编译时仅知道 T 是引用类型，因此必须使用对所有引用类型都有效的默认运算符。如果必须测试值相等性，建议的方法是同时应用 where T : IComparable<T> 约束，并在将用于构造泛型类的任何类中实现该接口。

约束多个参数

可以对多个参数应用约束，并对一个参数应用多个约束，如下面的示例所示：

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new() { }
```

未绑定的类型参数

没有约束的类型参数（如公共类 SampleClass<T>{} 中的 T）称为未绑定的类型参数。未绑定的类型参数具有以下规则：

- 不能使用 **!=** 和 **==** 运算符，因为无法保证具体类型参数能支持这些运算符。
- 可以在它们与 **System.Object** 之间来回转换，或将它们显式转换为任何接口类型。

- 可以将它们与 null 进行比较。将未绑定的参数与 **null** 进行比较时，如果类型参数为值类型，则该比较将始终返回 false。

作为约束的类型参数

将泛型类型参数作为约束使用，在具有自己类型参数的成员函数必须将该参数约束为包含类型的类型参数时非常有用，如下示例所示：

```
class List<T>
{
    void Add<U>(List<U> items) where U : T {/*...*/}
}
```

在上面的示例中，T 在 **Add** 方法的上下文中是一个类型约束，而在 **List** 类的上下文中是一个未绑定的类型参数。

类型参数还可在泛型类定义中用作约束。请注意，必须在尖括号中声明此类型参数与任何其他类型的参数：

```
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

泛型类的类型参数约束的作用非常有限，因为编译器除了假设类型参数派生自 **System.Object** 以外，不会做其他任何假设。在希望强制两个类型参数之间的继承关系的情况下，可对泛型类使用参数类型约束。

请参阅

System.Collections.Generic

C# 编程指南

泛型介绍（C# 编程指南）

泛型类（C# 编程指南）

new 约束（C# 参考）

泛型类（C# 编程指南）

泛型类封装不是特定于具体数据类型的操作。泛型类最常用于集合，如链接列表、哈希表、堆栈、队列、树等。像从集合中添加和移除项这样的操作都以大体上相同的方式执行，与所存储数据的类型无关。

对于大多数需要集合类的方案，推荐的方法是使用 .NET Framework 类库中所提供的类。有关使用这些类的更多信息，请参见.NET Framework 类库中的泛型（C# 编程指南）。

一般情况下，创建泛型类的过程为：从一个现有的具体类开始，逐一将每个类型更改为类型参数，直至达到通用化和可用性的最佳平衡。创建您自己的泛型类时，需要特别注意以下事项：

- 将哪些类型通用化为类型参数。

  通常，能够参数化的类型越多，代码就会变得越灵活，重用性就越好。但是，太多的通用化会使其他开发人员难以阅读或理解代码。

- 如果存在约束，应对类型参数应用什么约束（请参见 类型参数的约束（C# 编程指南））。

  一条有用的规则是，应用尽可能最多的约束，但仍使您能够处理必须处理的类型。例如，如果您知道您的泛型类仅用于引用类型，则应用类约束。这可以防止您的类被意外地用于值类型，并允许您对 T 使用 **as** 运算符以及检查空值。

- 是否将泛型行为分解为基类和子类。

  由于泛型类可以作为基类使用，此处适用的设计注意事项与非泛型类相同。请参见本主题后面有关从泛型基类继承的规则。

- 是否实现一个或多个泛型接口。

  例如，如果您设计一个类，该类将用于创建基于泛型的集合中的项，则可能必须实现一个接口，如 IComparable<T>，其中 T 是您的类的类型。

有关简单泛型类的示例，请参见泛型介绍（C# 编程指南）。

类型参数和约束的规则对于泛型类行为有几方面的含义，特别是关于继承和成员可访问性。您应当先理解一些术语，然后再继续进行。对于泛型类 Node<T>，客户端代码可通过指定类型参数来引用该类，以便创建封闭式构造类型 (Node<int>)。或者可以让类型参数处于未指定状态（例如在指定泛型基类时）以创建开放式构造类型 (Node<T>)。泛型类可以从具体的、封闭式构造或开放式构造基类继承：

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

非泛型类（换句话说，即具体类）可以从封闭式构造基类继承，但无法从开放式构造类或类型参数继承，因为在运行时客户端代码无法提供实例化基类所需的类型参数。

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

从开放式构造类型继承的泛型类必须为任何未被继承类共享的基类类型参数提供类型变量，如以下代码所示：

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

从开放式构造类型继承的泛型类必须指定约束，这些约束是基类型约束的超集或暗示基类型约束：

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable
```

泛型类型可以使用多个类型参数和约束，如下所示：

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

开放式构造类型和封闭式构造类型可以用作方法参数：

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

如果某个泛型类实现了接口，则可以将该类的所有实例强制转换为该接口。

泛型类是不变的。也就是说，如果输入参数指定 List<BaseClass>，则当您尝试提供 List<DerivedClass> 时，将会发生编译时错误。

请参阅

System.Collections.Generic

C# 编程指南

泛型（C# 编程指南）

Saving the State of Enumerators（保存枚举器的状态）

An Inheritance Puzzle, Part One（一个继承难题，第一部分）

泛型接口（C# 编程指南）

为泛型集合类或表示集合中项的泛型类定义接口通常很有用。对于泛型类，使用泛型接口十分可取，例如使用 IComparable<T> 而不使用 IComparable，这样可以避免值类型的装箱和取消装箱操作。.NET Framework 类库定义了若干泛型接口，以用于 System.Collections.Generic 命名空间中的集合类。

将接口指定为类型参数的约束时，只能使用实现此接口的类型。下面的代码示例显示从 SortedList<T> 类派生的 GenericList<T> 类。有关更多信息，请参见 泛型介绍（C# 编程指南）。 SortedList<T> 添加约束 where T : IComparable<T>。这将使 SortedList<T> 中的 **BubbleSort** 方法能够对列表元素使用泛型 CompareTo 方法。在此示例中，列表元素为简单类，即实现 Person 的 IComparable<Person>。

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerabl
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data;  //T as private member datatype

        public Node(T t)  //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        public T Data  //T as return type of property
        {
            get { return data; }
            set { data = value; }
        }
    }

    public GenericList()  //constructor
    {
        head = null;
    }
```

```
    public void AddHead(T t)  //T as method parameter type
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    // Implementation of the iterator
    public System.Collections.Generic.IEnumerator<T> GetEnumerator(
    {
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    // IEnumerable<T> inherits from IEnumerable, therefore this cla
    // must implement both the generic and non-generic versions of
    // GetEnumerator. In most cases, the non-generic method can
    // simply call the generic method.
    System.Collections.IEnumerator System.Collections.IEnumerable.G
    {
        return GetEnumerator();
    }
}

public class SortedList<T> : GenericList<T> where T : System.ICompa
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

        do
        {
            Node previous = null;
            Node current = head;
            swapped = false;

            while (current.next != null)
            {
                //  Because we need to call this method, the Sortec
                //  class is constrained on IEnumerable<T>
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
```

```
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
                    current = current.next;
                }
            }
        } while (swapped);
    }
}

// A simple class that implements IComparable<T> using itself as th
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
        return age - p.age;
    }

    public override string ToString()
    {
        return name + ":" + age;
    }

    // Must implement Equals.
    public bool Equals(Person p)
    {
```

```
            return (this.age == p.age);
        }
    }

    class Program
    {
        static void Main()
        {
            //Declare and instantiate a new generic SortedList class.
            //Person is the type argument.
            SortedList<Person> list = new SortedList<Person>();

            //Create name and age values to initialize Person objects.
            string[] names = new string[]
            {
                "Franscoise",
                "Bill",
                "Li",
                "Sandra",
                "Gunnar",
                "Alok",
                "Hiroyuki",
                "Maria",
                "Alessandro",
                "Raul"
            };

            int[] ages = new int[] { 45, 19, 28, 23, 18, 9, 108, 72, 3(

            //Populate the list.
            for (int x = 0; x < 10; x++)
            {
                list.AddHead(new Person(names[x], ages[x]));
            }

            //Print out unsorted list.
            foreach (Person p in list)
            {
                System.Console.WriteLine(p.ToString());
            }
            System.Console.WriteLine("Done with unsorted list");

            //Sort the list.
            list.BubbleSort();

            //Print out sorted list.
            foreach (Person p in list)
            {
                System.Console.WriteLine(p.ToString());
            }
            System.Console.WriteLine("Done with sorted list");
        }
    }
```

可将多重接口指定为单个类型上的约束，如下所示：

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

一个接口可定义多个类型参数，如下所示：

```
interface IDictionary<K, V>
{
}
```

适用于类的继承规则同样适用于接口：

```
interface IMonth<T> { }

interface IJanuary     : IMonth<int> { }  //No error
interface IFebruary<T> : IMonth<int> { }  //No error
interface IMarch<T>    : IMonth<T> { }    //No error
//interface IApril<T>  : IMonth<T, U> {}  //Error
```

如果泛型接口为逆变的，即仅使用其类型参数作为返回值，则此泛型接口可以从非泛型接口继承。在 .NET Framework 类库中，IEnumerable<T> 从 IEnumerable 继承，因为 IEnumerable<T> 只在 GetEnumerator 的返回值和 Current 属性 getter 中使用 T。

具体类可以实现已关闭的构造接口，如下所示：

```
interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }
```

只要类参数列表提供了接口必需的所有参数，泛型类便可以实现泛型接口或已关闭的构造接口，如下所示：

```
interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }          //No error
class SampleClass2<T> : IBaseInterface2<T, string> { }  //No error
```

对于泛型类、泛型结构或泛型接口中的方法，控制方法重载的规则相同。有关更多信息，请参见 泛型方法（C# 编程指南）。

请参阅

C# 编程指南

泛型介绍（C# 编程指南）

接口（C# 参考）

.NET Framework 中的泛型

泛型方法（C# 编程指南）

泛型方法是使用类型参数声明的方法，如下所示：

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

下面的代码示例演示一种使用 **int** 作为类型参数的方法调用方式：

```
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

也可以省略类型参数，编译器将推断出该参数。下面对 Swap 的调用等效于前面的调用：

```
Swap(ref a, ref b);
```

相同的类型推理规则也适用于静态方法和实例方法。编译器能够根据传入的方法实参推断类型形参；它无法仅从约束或返回值推断类型形参。因此，类型推理不适用于没有参数的方法。类型推理在编译时、编译器尝试解析重载方法签名之前进行。编译器向共享相同名称的所有泛型方法应用类型推理逻辑。在重载解析步骤中，编译器仅包括类型推理取得成功的那些泛型方法。

在泛型类中，非泛型方法可以访问类级别类型参数，如下所示：

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

如果定义采用相同类型参数作为包含类的泛型方法，编译器将生成警告 CS0693，因为在方法范围内为内部 T 提供的参数隐藏了为外部 T 提供的参数。如果需要使用其他类型参数（而不是实例化类时提供的类型参数）来灵活地调用泛型类方法，请考虑为方法的类型参数提供另一个标识符，如下面示例的 GenericList2<T> 中所示。

```
class GenericList<T>
{
    // CS0693
    void SampleMethod<T>() { }
}

class GenericList2<T>
{
    //No warning
    void SampleMethod<U>() { }
}
```

使用约束对方法中的类型参数启用更专门的操作。此版本的 Swap<T> 现在名为 SwapIfGreater<T>，它只能与实现 IComparable<T> 的类型参数一起使用。

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.ICompa
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}
```

泛型方法可以使用许多类型参数进行重载。例如，下列方法可以全部位于同一个类中：

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

C# 语言规范

有关更多信息，请参见 C# 语言规范。

请参阅

System.Collections.Generic

C# 编程指南

泛型介绍（C# 编程指南）

方法（C# 编程指南）

泛型和数组（C# 编程指南）

在 C# 2.0 以及更高版本中，下限为零的一维数组自动实现 IList<T>。这使您可以创建能够使用相同代码循环访问数组和其他集合类型的泛型方法。此技术主要对读取集合中的数据很有用。 IList<T> 接口不能用于在数组中添加或移除元素。如果尝试对此上下文中的数组调用 IList<T> 方法（例如 RemoveAt），则将引发异常。

下面的代码示例演示带有 IList<T> 输入参数的单个泛型方法如何同时循环访问列表和数组，本例中为整数数组。

```
class Program
{
    static void Main()
    {
        int[] arr = { 0, 1, 2, 3, 4 };
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item.ToString() + " ");
        }
        System.Console.WriteLine();
    }
}
```

请参阅

System.Collections.Generic

C# 编程指南

泛型（C# 编程指南）

数组（C# 编程指南）

.NET Framework 中的泛型

泛型委托（C# 编程指南）

委托 可以定义自己的类型参数。引用泛型委托的代码可以指定类型参数以创建已关闭的构造类型，就像实例化泛型类或调用泛型方法一样，如下例所示：

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

C# 2.0 版具有称为方法组转换的新功能，此功能适用于具体委托类型和泛型委托类型，并使您可以使用如下简化的语法写入上一行：

```
Del<int> m2 = Notify;
```

在泛型类内部定义的委托使用泛型类类型参数的方式可以与类方法所使用的方式相同。

```
class Stack<T>
{
    T[] items;
    int index;

    public delegate void StackDelegate(T[] items);
}
```

引用委托的代码必须指定包含类的类型变量，如下所示：

```
private static void DoWork(float[] items) { }

public static void TestStack()
{
    Stack<float> s = new Stack<float>();
    Stack<float>.StackDelegate d = DoWork;
}
```

根据典型设计模式定义事件时，泛型委托尤其有用，因为发送方参数可以为强类型，不再需要强制转换成 Object，或反向强制转换。

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs> stackE

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        stackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.Stack
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.stackEvent += o.HandleStackChange;
}
```

请参阅

System.Collections.Generic

C# 编程指南

泛型介绍（C# 编程指南）

泛型方法（C# 编程指南）

泛型类（C# 编程指南）

泛型接口（C# 编程指南）

委托（C# 编程指南）

.NET Framework 中的泛型

泛型代码中的默认关键字（C# 编程指南）

在泛型类和泛型方法中产生的一个问题是，在预先未知以下情况时，如何将默认值分配给参数化类型 T：

- T 是引用类型还是值类型。
- 如果 T 为值类型，则它是数值还是结构。

给定参数化类型 T 的一个变量 t，只有当 T 为引用类型时，语句 t = null 才有效；只有当 T 为数值类型而不是结构时，语句 t = 0 才能正常使用。解决方案是使用 **default** 关键字，此关键字对于引用类型会返回 null，对于数值类型会返回零。对于结构，此关键字将返回初始化为零或 null 的每个结构成员，具体取决于这些结构是值类型还是引用类型。对于可以为 null 的值类型，默认返回 System.Nullable<T>，它像任何结构一样初始化。

以下来自 GenericList<T> 类的示例显示了如何使用 **default** 关键字。有关更多信息，请参见泛型概述。

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Test with a non-empty list of integers.
            GenericList<int> gll = new GenericList<int>();
            gll.AddNode(5);
            gll.AddNode(4);
            gll.AddNode(3);
            int intVal = gll.GetLast();
            // The following line displays 5.
            System.Console.WriteLine(intVal);

            // Test with an empty list of integers.
            GenericList<int> gll2 = new GenericList<int>();
            intVal = gll2.GetLast();
            // The following line displays 0.
            System.Console.WriteLine(intVal);

            // Test with a non-empty list of strings.
            GenericList<string> gll3 = new GenericList<string>();
            gll3.AddNode("five");
            gll3.AddNode("four");
            string sVal = gll3.GetLast();
            // The following line displays five.
            System.Console.WriteLine(sVal);

            // Test with an empty list of strings.
```

```
            GenericList<string> gll4 = new GenericList<string>();
            sVal = gll4.GetLast();
            // The following line displays a blank line.
            System.Console.WriteLine(sVal);
        }
    }

    // T is the type of data stored in a particular instance of Ger
    public class GenericList<T>
    {
        private class Node
        {
            // Each node has a reference to the next node in the li
            public Node Next;
            // Each node holds a value of type T.
            public T Data;
        }

        // The list is initially empty.
        private Node head = null;

        // Add a node at the beginning of the list with t as its da
        public void AddNode(T t)
        {
            Node newNode = new Node();
            newNode.Next = head;
            newNode.Data = t;
            head = newNode;
        }

        // The following method returns the data value stored in th
        // the list. If the list is empty, the default value for ty
        // returned.
        public T GetLast()
        {
            // The value of temp is returned as the value of the me
            // The following declaration initializes temp to the ap
            // default value for type T. The default value is retur
            // list is empty.
            T temp = default(T);

            Node current = head;
            while (current != null)
            {
                temp = current.Data;
                current = current.Next;
            }
            return temp;
        }
    }
}
```

请参阅

System.Collections.Generic

C# 编程指南

泛型（C# 编程指南）

泛型方法（C# 编程指南）

.NET Framework 中的泛型

C++ 模板和 C# 泛型之间的区别（C# 编程指南）

C# 泛型和 C++ 模板都是用于提供参数化类型支持的语言功能。 然而，这两者之间存在许多差异。 在语法层面上，C# 泛型是实现参数化类型的更简单方法，不具有 C++ 模板的复杂性。 此外，C# 并不尝试提供 C++ 模板所提供的所有功能。 在实现层面，主要区别在于，C# 泛型类型替换是在运行时执行的，从而为实例化的对象保留了泛型类型信息。 有关更多信息，请参见 运行时中的泛型（C# 编程指南）。

以下是 C# 泛型和 C++ 模板之间的主要差异：

- C# 泛型未提供与 C++ 模板相同程度的灵活性。 例如，尽管在 C# 泛型类中可以调用用户定义的运算符，但不能调用算术运算符。
- C# 不允许非类型模板参数，如 template C<int i> {}。
- C# 不支持显式专用化，即特定类型的模板的自定义实现。
- C# 不支持部分专用化：类型参数子集的自定义实现。
- C# 不允许将类型参数用作泛型类型的基类。
- C# 不允许类型参数具有默认类型。
- 在 C# 中，尽管构造类型可用作泛型，但泛型类型参数自身不能是泛型。 C++ 确实允许模板参数。
- C++ 允许那些可能并非对模板中的所有类型参数都有效的代码，然后将检查该代码中是否有用作类型参数的特定类型。 C# 要求相应地编写类中的代码，使之能够使用任何满足约束的类型。 例如，可以在 C++ 中编写对类型参数的对象使用算术运算符 + 和 - 的函数，这会在使用不支持这些运算符的类型来实例化模板时产生错误。 C# 不允许这样；唯一允许的语言构造是那些可从约束推导出来的构造。

请参见

参考

泛型介绍（C# 编程指南）

概念

C# 编程指南

其他资源

模板

运行时中的泛型（C# 编程指南）

将泛型类型或方法编译为 Microsoft 中间语言 (MSIL) 时，它包含将其标识为具有类型参数的元数据。 泛型类型的 MSIL 的使用因所提供的类型参数是值类型还是引用类型而不同。

第一次用值类型作为参数来构造泛型类型时，运行时会创建专用泛型类型，将提供的参数代入到 MSIL 中的适当位置。 对于每个用作参数的唯一值类型，都会创建一次专用泛型类型。

例如，假设您的程序代码声明了一个由整数构造的堆栈：

```
Stack<int> stack;
```

在此位置，运行时生成 Stack<T> 类的专用版本，并相应地用整数替换其参数。 现在，只要程序代码使用整数堆栈，运行时就会重用生成的专用 Stack<T> 类。 在下面的示例中，创建了整数堆栈的两个实例，它们共享 Stack<int> 代码的单个实例：

```
Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

但是，假定在代码中的另一个位置创建了使用不同值类型（比如 long 或用户定义的结构）作为其参数的另一个 Stack<T> 类。 因此，运行时将生成另一个版本的泛型类型，并在 MSIL 中的适当位置替换 long。 由于每个专用泛型类本身就包含值类型，因此不再需要转换。

对于引用类型，泛型的工作方式略有不同。 第一次使用任何引用类型构造泛型类型时，运行时会创建专用泛型类型，用对象引用替换 MSIL 中的参数。 然后，每次使用引用类型作为参数来实例化构造类型时，无论引用类型的具体类型是什么，运行时都会重用以前创建的泛型类型的专用版本。 之所以可以这样，是因为所有引用的大小相同。

例如，假设您有两个引用类型：一个 Customer 类和一个 Order 类，并且同时假设您创建了一个 Customer 类型的堆栈：

```
class Customer { }
class Order { }
```

```
Stack<Customer> customers;
```

此时，运行时将生成 Stack<T> 类的一个专用版本，该版本存储稍后将填写的对象引用，而不是存储数据。 假设下一行代码创建另一个引用类型的堆栈，该堆栈名为 Order：

```
Stack<Order> orders = new Stack<Order>();
```

不同于值类型，对于 Order 类型不创建 Stack<T> 类的另一个专用版本。 而是创建 Stack<T> 类的一个专用版本实例，并将 orders 变量设置为引用它。 假设接下来您遇到一行创建 Customer 类型堆栈的代码：

```
customers = new Stack<Customer>();
```

与前面使用 Order 类型创建的 Stack<T> 类一样，创建了专用 Stack<T> 类的另一个实例。 包含在其中的指针设置为引用 Customer 类型大小的内存区域。 因为引用类型的数量会随程序的不同而大幅变化，C# 泛型实现将编译器为引用类型的泛型类创建的专用类的数量减小到一个，从而大幅减小代码量。

此外，使用值类型或引用类型参数实例化泛型 C# 类时，反射可以在运行时查询它，并且可以确定它的实际类型及其类型参数。

请参见

参考

泛型介绍（C# 编程指南） System.Collections.Generic

概念

C# 编程指南

其他资源

.NET Framework 中的泛型

.NET Framework 类库中的泛型（C# 编程指南）

.NET Framework 2.0 版类库提供一个新的命名空间 System.Collections.Generic，其中包括几个随时可用的泛型集合类和关联接口。其他命名空间（如 System）也提供新的泛型接口，如 IComparable<T>。与早期版本的 .NET Framework 所提供的非泛型集合类相比，这些类和接口更为高效和类型安全。在设计和实现自己的自定义集合类之前，请考虑是否能够使用 .NET Framework 类库所提供的类，或者是否能从 .NET Framework 类库所提供的类派生一个类。

请参阅

C# 编程指南

何时使用泛型集合

集合和数据结构

泛型介绍（C# 编程指南）

泛型和反射（C# 编程指南）

因为公共语言运行时 (CLR) 能够在运行时访问泛型类型信息，所以可以使用反射获取关于泛型类型的信息，方法与用于非泛型类型的方法相同。有关更多信息，请参见 运行时中的泛型（C# 编程指南）。

在 .NET Framework 2.0 中，有几个新成员添加到了 Type 类中，用以启用泛型类型的运行时信息。请参见有关这些类的文档来了解有关如何使用这些方法和属性的更多信息。System.Reflection.Emit 命名空间还包含支持泛型的新成员。请参见 如何：用反射发出定义泛型类型。

有关泛型反射中使用的术语的固定条件列表，请参见 IsGenericType 属性备注。

System.Type 成员名称	说明
IsGenericType	如果类型为泛型，则返回 true。
GetGenericArguments	返回 **Type** 对象数组，这些对象表示为构造类型提供的类型变量，或泛型类型定义的类型参数。
GetGenericTypeDefinition	返回当前构造类型的基础泛型类型定义。
GetGenericParameterConstraints	返回表示当前泛型类型参数约束的 **Type** 对象的数组。
ContainsGenericParameters	如果类型或其任意封闭类型或方法包含没有被提供特定类型的类型参数，则返回 true。
GenericParameterAttributes	获取 **GenericParameterAttributes** 标志的组合，这些标志描述当前泛型类型参数的特殊约束。
GenericParameterPosition	对于表示类型参数的 **Type** 对象，获取类型参数在声明该类型参数的泛型类型定义或泛型方法定义的类型参数列表中的位置。
IsGenericParameter	获取一个值，该值指示当前 **Type** 是表示泛型类型定义的类型参数，还是泛型方法定义的类型参数。
IsGenericTypeDefinition	获取一个值，该值指示当前 Type 是否表示可以用来构造其他泛型类型的泛型类型定义。如果类型表示泛型类型的定义，则返回 true。
DeclaringMethod	返回定义当前泛型类型参数的泛型方法；如果类型参数不是由泛型方法定义的，则返回空值。
MakeGenericType	用类型数组的元素替代当前泛型类型定义的类型参数，并返回表示结果构造类型的 Type 对象。

此外，MethodInfo 类中还添加了新成员以启用泛型方法的运行时信息。有关泛型方法反射中使用的术语的固定条件列表，请参见 IsGenericMethod 属性备注。

System.Reflection.MemberInfo 成员名称	说明
IsGenericMethod	如果方法为泛型，则返回 true。
GetGenericArguments	返回 Type 对象数组，这些对象表示构造泛型方法的类型变量，或泛型方法定义的类型参数。
GetGenericMethodDefinition	返回当前构造方法的基础泛型方法定义。
ContainsGenericParameters	如果方法或其任意封闭类型包含没有被提供特定类型的任何类型参数，则返回 true。
IsGenericMethodDefinition	如果当前 MethodInfo 表示泛型方法的定义，则返回 true。
MakeGenericMethod	用类型数组的元素替代当前泛型方法定义的类型参数，并返回表示结果构造方法的 MethodInfo 对象。

请参阅

C# 编程指南

泛型（C# 编程指南）

反射类型和泛型类型

.NET Framework 中的泛型

泛型和特性（C# 编程指南）

特性可以应用于泛型类型中，方式与应用于非泛型类型相同。有关应用特性的更多信息，请参见 特性（C# 和 Visual Basic）。

自定义特性只允许引用开放泛型类型（未提供类型参数的泛型类型）和封闭构造泛型类型（为所有类型参数提供参数）。

下面的示例使用此自定义特性：

```
class CustomAttribute : System.Attribute
{
    public System.Object info;
}
```

特性可以引用开放式泛型类型：

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

使用数目适当的若干个逗号指定多个类型参数。在此示例中，GenericClass2 有两个类型参数：

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<,>))]
class ClassB { }
```

特性可以引用封闭式构造泛型类型：

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

引用泛型类型参数的特性将导致编译时错误：

```
//[CustomAttribute(info = typeof(GenericClass3<int, T, string>))]
class ClassD<T> { }
```

不能从 Attribute 继承泛型类型：

```
//public class CustomAtt<T> : System.Attribute {}  //Error
```

若要在运行时获得有关泛型类型或类型参数的信息，可以使用 System.Reflection 的方法。有关更多信息，请参见泛型和反射（C# 编程指南）

请参阅

C# 编程指南

泛型（C# 编程指南）

利用特性扩展元数据

索引器（C# 编程指南）

索引器允许类或结构的实例就像数组一样进行索引。索引器类似于属性，不同之处在于它们的取值函数采用参数。

在下面的示例中，定义了一个泛型类，并为其提供了简单的 get 和 set 取值函数方法（作为分配和检索值的方法）。 Program 类创建了此类的一个实例，用于存储字符串。

```
class SampleCollection<T>
{
    // Declare an array to store the data elements.
    private T[] arr = new T[100];

    // Define the indexer, which will allow client code
    // to use [] notation on the class instance itself.
    // (See line 2 of code in Main below.)
    public T this[int i]
    {
        get
        {
            // This indexer is very simple, and just returns or set
            // the corresponding element from the internal array.
            return arr[i];
        }
        set
        {
            arr[i] = value;
        }
    }
}

// This class shows how client code uses the indexer.
class Program
{
    static void Main(string[] args)
    {
        // Declare an instance of the SampleCollection type.
        SampleCollection<string> stringCollection = new SampleColle

        // Use [] notation on the type.
        stringCollection[0] = "Hello, World";
        System.Console.WriteLine(stringCollection[0]);
    }
}
// Output:
// Hello, World.
```

> 注意
>
> 有关更多示例，请参阅相关章节。

表达式主体定义

直接只返回表达式结果的索引器很常见。下面的语法快捷方式使用 **=>** 来定义这些索引器：

```
public Customer this[long id] => store.LookupCustomer(id);
```

索引器必须为只读，并且你不能使用 **get** 取值函数关键字。

索引器概述

- 使用索引器可以用类似于数组的方式为对象建立索引。
- **get** 取值函数返回值。 **set** 取值函数分配值。
- this 关键字用于定义索引器。
- value 关键字用于定义由 **set** 索引器分配的值。
- 索引器不必根据整数值进行索引；由你决定如何定义特定的查找机制。
- 索引器可被重载。
- 索引器可以有多个形参，例如当访问二维数组时。

相关章节

- 使用索引器（C# 编程指南）
- 接口中的索引器（C# 编程指南）
- 属性和索引器之间的比较（C# 编程指南）
- 限制访问器可访问性（C# 编程指南）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

属性（C# 编程指南）

使用索引器（C# 编程指南）

索引器在语法上方便您创建客户端应用程序可将其作为数组访问的类、结构或接口。索引器经常是在主要用于封装内部集合或数组的类型中实现的。例如，假定具有一个名为 TempRecord 的类，此类表示在 24 小时内的 10 个不同时间记录的华氏度。此类包含一个表示温度的 float 类型的名为“temps”的数组和表示记录温度的日期的 DateTime。通过在此类中实现一个索引器，客户端可以通过 float temp = tr[4] 而不是 float temp = tr.temps[4] 语法访问 TempRecord 实例中的温度。索引器表示法不仅简化了客户端应用程序的语法，还使其他开发人员能够更加直观地理解类及其用途。

要声明类或结构上的索引器，请使用 this 关键字，如下例所示：

```
public int this[int index]    // Indexer declaration
{
    // get and set accessors
}
```

备注

索引器类型及其参数类型必须至少如同索引器本身一样是可访问的。有关可访问级别的更多信息，请参见访问修饰符。

有关如何对接口使用索引器的更多信息，请参见接口索引器。

索引器的签名由其形参的数量和类型组成。它不包括索引器类型或形参名。如果在同一类中声明一个以上的索引器，则它们必须具有不同的签名。

索引器值不属于变量；因此，不能将索引器值作为 ref 或 out 参数进行传递。

要为索引器提供一个其他语言可以使用的名字，请使用声明中的 name 特性。例如：

```
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this [int index]   // Indexer declaration
{
}
```

此索引器将具有名称 TheItem。不提供名称特性将生成 Item 默认名称。

示例 1

说明

下面的示例说明如何声明私有数组字段、temps 和索引器。使用索引器可直接访问实例 tempRecord[i]。另一种使用索引器的方法是将数组声明为 public 成员并直接访问它的成员 tempRecord.temps[i]。

请注意，当计算索引器的访问时（例如，在 **Console.Write** 语句中），将调用 get 访问器。因此，如果 **get** 访问器不存在，将发生编译时错误。

代码

```
class TempRecord
{
    // Array of temperature values
    private float[] temps = new float[10] { 56.2F, 56.7F, 56.5F, 56
                                            61.3F, 65.9F, 62.1F, 59

    // To enable client code to validate input
    // when accessing your indexer.
    public int Length
    {
        get { return temps.Length; }
    }
    // Indexer declaration.
    // If index is out of range, the temps array will throw the exc
    public float this[int index]
    {
        get
        {
            return temps[index];
        }

        set
        {
            temps[index] = value;
        }
    }
}

class MainClass
{
    static void Main()
    {
        TempRecord tempRecord = new TempRecord();
        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
            System.Console.WriteLine("Element #{0} = {1}", i, tempR
        }
```

```
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();

    }
}
/* Output:
        Element #0 = 56.2
        Element #1 = 56.7
        Element #2 = 56.5
        Element #3 = 58.3
        Element #4 = 58.8
        Element #5 = 60.1
        Element #6 = 65.9
        Element #7 = 62.1
        Element #8 = 59.2
        Element #9 = 57.5
    */
```

使用其他值进行索引

C# 并不将索引类型限制为整数。例如，对索引器使用字符串可能是有用的。通过搜索集合内的字符串并返回相应的值，可以实现此类索引器。由于访问器可被重载，字符串和整数版本可以共存。

示例 2

说明

在此例中，声明了存储星期几的类。声明了一个 **get** 访问器，它接受字符串（天名称），并返回相应的整数。例如，星期日将返回 0，星期一将返回 1，等等。

代码

```
// Using a string as an indexer value
class DayCollection
{
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri",

    // This method finds the day or returns -1
    private int GetDay(string testDay)
    {

        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == testDay)
            {
                return j;
            }
        }

        throw new System.ArgumentOutOfRangeException(testDay, "test
    }

    // The get accessor returns an integer for a given string
    public int this[string day]
    {
        get
        {
            return (GetDay(day));
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        DayCollection week = new DayCollection();
        System.Console.WriteLine(week["Fri"]);

        // Raises ArgumentOutOfRangeException
        System.Console.WriteLine(week["Made-up Day"]);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
// Output: 5
```

可靠编程

提高索引器的安全性和可靠性有两种主要的方法：

- 确保结合某一类型的错误处理策略，以处理万一客户端代码传入无效索引值的情况。在本主题前面的第一个示例中，TempRecord 类提供了 Length 属性，使客户端代码能够在将输入传递给索引器之前对其进行验证。也可以将错误处理代码放入索引器自身内部。确保为用户记录在索引器的访问器中引发的任何异常。
- 应当为 **get** 和 set 访问器的可访问性设置尽可能多的限制。这一点对 **set** 访问器尤为重要。有关更多信息，请参见 限制访问器可访问性（C# 编程指南）。

请参阅

C# 编程指南

索引器（C# 编程指南）

属性（C# 编程指南）

接口中的索引器（C# 编程指南）

索引器可在 接口（C# 参考） 上声明。接口索引器的访问器与类索引器的访问器具有以下方面的不同：

- 接口访问器不使用修饰符。
- 接口访问器没有体。

因此，访问器的用途是指示索引器是读写、只读还是只写。

以下是接口索引器访问器的示例：

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

一个索引器的签名必须区别于在同一接口中声明的其他所有索引器的签名。

下面的示例显示如何实现接口索引器。

```
// Indexer on an interface:
public interface ISomeInterface
{
    // Indexer declaration:
    int this[int index]
    {
        get;
        set;
    }
}

// Implementing the interface.
class IndexerClass : ISomeInterface
{
    private int[] arr = new int[100];
    public int this[int index]   // indexer declaration
    {
        get
        {
```

```
            // The arr object will throw IndexOutOfRange exception.
            return arr[index];
        }
        set
        {
            arr[index] = value;
        }
    }
}

class MainClass
{
    static void Main()
    {
        IndexerClass test = new IndexerClass();
        System.Random rand = new System.Random();
        // Call the indexer to initialize its elements.
        for (int i = 0; i < 10; i++)
        {
            test[i] = rand.Next();
        }
        for (int i = 0; i < 10; i++)
        {
            System.Console.WriteLine("Element #{0} = {1}", i, test[

        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Sample output:
    Element #0 = 360877544
    Element #1 = 327058047
    Element #2 = 1913480832
    Element #3 = 1519039937
    Element #4 = 601472233
    Element #5 = 323352310
    Element #6 = 1422639981
    Element #7 = 1797892494
    Element #8 = 875761049
    Element #9 = 393083859
 */
```

在上例中，可以通过使用接口成员的完全限定名来使用显式接口成员实现。例如：

```
public string ISomeInterface.this
{
}
```

但是，只有当类使用同一索引器签名实现一个以上的接口时，为避免多义性才需要使用完全限定名。例如，如果 Employee 类实现的是两个接口 ICitizen 和 IEmployee，并且这两个接口具有相同的索引器签名，则必须使用显式接口成员实现。即，以下索引器声明：

```
public string IEmployee.this
{
}
```

在 IEmployee 接口上实现索引器，而以下声明：

```
public string ICitizen.this
{
}
```

在 ICitizen 接口上实现索引器。

请参阅

C# 编程指南

索引器（C# 编程指南）

属性（C# 编程指南）

接口（C# 编程指南）

属性和索引器之间的比较（C# 编程指南）

索引器与属性类似。除下表中显示的差别外，为属性访问器定义的所有规则同样适用于索引器访问器。

属性	索引器
允许像调用公共数据成员一样调用方法。	允许对一个对象本身使用数组表示法来访问该对象内部集合中的元素。
可通过简单的名称进行访问。	可通过索引器进行访问。
可以为静态成员或实例成员。	必须为实例成员。
属性的 get 访问器没有参数。	索引器的 **get** 访问器具有与索引器相同的形参表。
属性的 set 访问器包含隐式 **value** 参数。	除了值参数外，索引器的 **set** 访问器还具有与索引器相同的形参表。
支持对自动实现的属性（C# 编程指南）使用短语法。	不支持短语法。

请参阅

C# 编程指南

索引器（C# 编程指南）

属性（C# 编程指南）

接口（C# 编程指南）

接口包含类或结构可以实现的一组相关功能的定义。

例如，使用接口可以在类中包括来自多个源的行为。 由于C#语言不支持多重继承，所以该功能很重要。 此外，如果要模拟结构的继承，也必须使用接口，因为它们无法实际从另一个结构或类继承。

可使用 interface 关键字定义接口，如以下示例所示。

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

实现 IEquatable<T> 接口的任何类或结构都必须包含与该接口指定的签名匹配的 Equals 方法的定义。因此，可以依靠实现 IEquatable<T> 的类来包含 Equals 方法，类的实例可以通过该方法确定它是否等于相同类的另一个实例。

IEquatable<T> 的定义不为 Equals 提供实现。该接口仅定义签名。这样，C# 中的接口便类似于其中所有方法都是抽象方法的抽象类。但是，类或结构可以实现多个接口，但是类只能继承单个类（抽象或不抽象）。因此，使用接口可以在类中包括来自多个源的行为。

有关抽象类的详细信息，请参阅抽象类、密封类及类成员。

接口可以包含方法、属性、事件、索引器或这四种成员类型的任意组合。有关示例的链接，请参阅相关章节。接口不能包含常量、字段、运算符、实例构造函数、析构函数或类型。接口成员会自动成为公共成员，不能包含任何访问修饰符。成员也不能是静态成员。

若要实现接口成员，实现类的对应成员必须是公共、非静态，并且具有与接口成员相同的名称和签名。

当类或结构实现接口时，类或结构必须为该接口定义的所有成员提供实现。接口本身不提供类或结构可以通过继承基类功能的方式来继承的任何功能。但是，如果基类实现接口，则从基类派生的任何类都会继承该实现。

下面的示例演示 IEquatable<T> 接口的实现。实现类 Car 必须提供 Equals 方法的实现。

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        if (this.Make == car.Make &&
            this.Model == car.Model &&
            this.Year == car.Year)
        {
            return true;
        }
        else
            return false;
    }
}
```

类的属性和索引器可以为接口中定义的属性或索引器定义额外的访问器。例如，一个接口可能会声明一个具有 get 访问器的属性。实现该接口的类可以声明同时具有 **get** 和 set 访问器的相同属性。但是，如果属性或索引器使用显式实现，则访问器必须匹配。有关显式实现的详细信息，请参阅显式接口实现（C# 编程指南）和接口属性（C# 编程指南）。

接口可以实现其他接口。类可能通过它继承的基类或通过其他接口实现的接口来多次包含某个接口。但是，类只能提供接口的实现一次，并且仅当类将接口作为类定义的一部分 (class ClassName : InterfaceName) 进行声明时才能提供。如果由于继承实现接口的基类而继承了接口，则基类会提供接口的成员的实现。但是，派生类可以重新实现接口成员，而不是使用继承的实现。

基类还可以使用虚拟成员实现接口成员。在这种情况下，派生类可以通过重写虚拟成员来更改接口行为。有关虚拟成员的详细信息，请参阅多态性。

接口摘要

接口具有以下属性：

- 接口类似于抽象基类。实现接口的任何类或结构都必须实现其所有成员。
- 接口无法直接进行实例化。其成员由实现接口的任何类或结构来实现。
- 接口可以包含事件、索引器、方法和属性。
- 接口不包含方法的实现。
- 一个类或结构可以实现多个接口。一个类可以继承一个基类，还可实现一个或多个接口。

本节内容

显式接口实现（C# 编程指南）

说明如何创建特定于接口的类成员。

如何：显式实现接口成员（C# 编程指南）

提供有关如何显式实现接口的成员的示例。

如何：显式实现两个接口的成员（C# 编程指南）

提供有关如何使用继承显式实现接口的成员的示例。

相关章节

- 接口属性（C# 编程指南）
- 接口中的索引器（C# 编程指南）
- 如何：实现接口事件（C# 编程指南）
- 类和结构（C# 编程指南）
- 继承（C# 编程指南）
- 方法（C# 编程指南）
- 多态性（C# 编程指南）
- 抽象类、密封类及类成员（C# 编程指南）
- 属性（C# 编程指南）
- 事件（C# 编程指南）
- 索引器（C# 编程指南）

重要章节

Learning C# 3.0: Master the Fundamentals of C# 3.0（学习 C# 3.0：掌握 C# 3.0 的基本知识）中的 Interfaces（接口）

请参阅

C# 编程指南

继承（C# 编程指南）

显式接口实现（C# 编程指南）

如果类实现两个接口，并且这两个接口包含具有相同签名的成员，那么在类中实现该成员将导致两个接口都使用该成员作为它们的实现。在下面的示例中，所有对 Paint 调用方法相同。

```
class Test
{
    static void Main()
    {
        SampleClass sc = new SampleClass();
        IControl ctrl = (IControl)sc;
        ISurface srfc = (ISurface)sc;

        // The following lines all call the same method.
        sc.Paint();
        ctrl.Paint();
        srfc.Paint();
    }
}

interface IControl
{
    void Paint();
}
interface ISurface
{
    void Paint();
}
class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}

// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

然而，如果两个接口成员执行不同的函数，那么这可能会导致其中一个接口的实现不正确或两个接口的实现都不正确。可以显式地实现接口成员 -- 即创建一个仅通过该接口调用并且特定于该接口的类成员。这是使用接口名称和一个句点命名该类成员来实现的。例如：

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

类成员 **IControl.Paint** 只能通过 **IControl** 接口使用，**ISurface.Paint** 只能通过 **ISurface** 使用。两个方法实现都是分离的，都不可以直接在类中使用。例如：

```
// Call the Paint methods from Main.

SampleClass obj = new SampleClass();
//obj.Paint();  // Compiler error.

IControl c = (IControl)obj;
c.Paint();  // Calls IControl.Paint on SampleClass.

ISurface s = (ISurface)obj;
s.Paint(); // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

显式实现还用于解决两个接口分别声明具有相同名称的不同成员（如属性和方法）的情况：

```
interface ILeft
{
    int P { get;}
}
interface IRight
{
    int P();
}
```

为了同时实现两个接口，类必须对属性 P 和/或方法 P 使用显式实现以避免编译器错误。例如：

```
class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

请参阅

C# 编程指南

类和结构（C# 编程指南）

接口（C# 编程指南）

继承（C# 编程指南）

如何：显式实现接口成员（C# 编程指南）

本示例声明一个 接口IDimensions 和一个类 Box，该类显式实现接口成员 getLength 和 getWidth。通过接口实例 dimensions 访问这些成员。

```
interface IDimensions
{
    float getLength();
    float getWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.getLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.getWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = (IDimensions)box1;

        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemen
        // interface member from a class instance:
        //System.Console.WriteLine("Length: {0}", box1.getLength()
        //System.Console.WriteLine("Width: {0}", box1.getWidth());

        // Print out the dimensions of the box by calling the metho
        // from an instance of the interface:
        System.Console.WriteLine("Length: {0}", dimensions.getLengt
```

```
            System.Console.WriteLine("Width: {0}", dimensions.getWidth(
        }
    }
    /* Output:
        Length: 30
        Width: 20
    */
```

可靠编程

- 请注意 **Main** 方法中下列代码行被注释掉，因为它们将产生编译错误。显式实现的接口成员不能从类实例访问：

  ```
  //System.Console.WriteLine("Length: {0}", box1.getLength());
  //System.Console.WriteLine("Width: {0}", box1.getWidth());
  ```

- 还请注意，**Main** 方法中的下列代码行成功输出框的尺寸，因为这些方法是从接口实例调用的：

  ```
  System.Console.WriteLine("Length: {0}", dimensions.getLength())
  System.Console.WriteLine("Width: {0}", dimensions.getWidth());
  ```

请参阅

C# 编程指南

类和结构（C# 编程指南）

接口（C# 编程指南）

如何：显式实现两个接口的成员（C# 编程指南）

如何：显式实现两个接口的成员（C# 编程指南）

显式接口实现还允许程序员实现具有相同成员名称的两个接口，并为每个接口成员各提供一个实现。本示例同时以公制单位和英制单位显示框的尺寸。Box 类实现 IEnglishDimensions 和 IMetricDimensions 两个接口，它们表示不同的度量系统。两个接口有相同的成员名称 Length 和 Width。

```
// Declare the English units interface:
interface IEnglishDimensions
{
    float Length();
    float Width();
}

// Declare the metric units interface:
interface IMetricDimensions
{
    float Length();
    float Width();
}

// Declare the Box class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float lengthInches;
    float widthInches;

    public Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }

    // Explicitly implement the members of IEnglishDimensions:
    float IEnglishDimensions.Length()
    {
        return lengthInches;
    }

    float IEnglishDimensions.Width()
    {
        return widthInches;
    }

    // Explicitly implement the members of IMetricDimensions:
    float IMetricDimensions.Length()
    {
        return lengthInches * 2.54f;
```

```
    }

    float IMetricDimensions.Width()
    {
        return widthInches * 2.54f;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an instance of the English units interface:
        IEnglishDimensions eDimensions = (IEnglishDimensions)box1;

        // Declare an instance of the metric units interface:
        IMetricDimensions mDimensions = (IMetricDimensions)box1;

        // Print dimensions in English units:
        System.Console.WriteLine("Length(in): {0}", eDimensions.Ler
        System.Console.WriteLine("Width (in): {0}", eDimensions.Wic

        // Print dimensions in metric units:
        System.Console.WriteLine("Length(cm): {0}", mDimensions.Ler
        System.Console.WriteLine("Width (cm): {0}", mDimensions.Wic
    }
}
/* Output:
    Length(in): 30
    Width (in): 20
    Length(cm): 76.2
    Width (cm): 50.8
*/
```

可靠编程

如果希望默认度量采用英制单位，请正常实现 Length 和 Width 这两个方法，并从 IMetricDimensions 接口显式实现 Length 和 Width 方法：

```
// Normal implementation:
public float Length()
{
    return lengthInches;
}
public float Width()
{
    return widthInches;
}

// Explicit implementation:
float IMetricDimensions.Length()
{
    return lengthInches * 2.54f;
}
float IMetricDimensions.Width()
{
    return widthInches * 2.54f;
}
```

这种情况下，可以从类实例访问英制单位，而从接口实例访问公制单位：

```
public static void Test()
{
    Box box1 = new Box(30.0f, 20.0f);
    IMetricDimensions mDimensions = (IMetricDimensions)box1;

    System.Console.WriteLine("Length(in): {0}", box1.Length());
    System.Console.WriteLine("Width (in): {0}", box1.Width());
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length(
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width()
}
```

请参阅

C# 编程指南

类和结构（C# 编程指南）

接口（C# 编程指南）

如何：显式实现接口成员（C# 编程指南）

互操作性（C# 编程指南）

互操作性使您能够保留和利用在现有非托管代码中的投入。 运行在公共语言运行时 (CLR) 的控制之下的代码称为“托管代码”，运行在 CLR 之外的代码称为“非托管代码”。 COM、COM+、C++ 组件、ActiveX 组件和 Microsoft Win32 API 都是非托管代码的示例。

.NET Framework 通过平台调用服务、System.Runtime.InteropServices 命名空间、C++ 互操作性和 COM 互操作性（COM 互操作）来实现与非托管代码的互操作性。

本节内容

互操作性概述（C# 编程指南）

介绍在 C# 托管代码和非托管代码之间进行互操作的方法。

如何：通过使用 Visual C# 功能访问 Office 互操作对象（C# 编程指南）

描述 Visual C# 2010 中引入的功能以便于 Office 编程。

如何：在 COM 互操作编程中使用索引属性（C# 编程指南）

描述如何使用索引属性以访问包含参数的 COM 属性。

如何：使用平台调用播放波形文件（C# 编程指南）

介绍如何使用平台调用服务在 Windows 操作系统中播放 .wav 声音文件。

演练：Office 编程（C# 和 Visual Basic）

演示如何创建包含一个指向该工作簿的 Excel 工作簿和 Word 文档。

COM 类示例（C# 编程指南）

演示如何将 C# 类作为 COM 对象公开。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参见

任务

演练：Office 编程（C# 和 Visual Basic）

参考

Marshal.ReleaseComObject

概念

C# 编程指南

其他资源

与非托管代码交互操作

互操作性概述（C# 编程指南）

本主题描述在 C# 托管和非托管代码之间实现互操作性的方法。

平台调用

平台调用是一种服务，它使托管代码可以调用在动态链接库 (DLL)（如 Microsoft Win32 API 中的那些 DLL）中实现的非托管函数。此服务将查找并调用导出的函数，然后根据需要跨越互用边界封送其参数（整数、字符串、数组、结构等）。

有关更多信息，请参见使用非托管 DLL 函数和 如何：使用平台调用播放波形文件（C# 编程指南）。

> 注意
>
> 公共语言运行时 (CLR) (CLR) 管理对系统资源的访问。调用 CLR 外部的非托管代码会避开此安全机制，因此会带来安全风险。例如，非托管代码可能会直接调用非托管代码中的资源，从而避开 CLR 安全机制，。有关更多信息，请参见 .NET Framework Security（.NET Framework 安全性）。

C++ Interop

可以使用 C++ Interop（又称为 It Just Works (IJW)）包装本机 C++ 类，使得用 C# 或其他 .NET Framework 语言编写的代码可以使用它。为此，可以编写 C++ 代码来包装本机 DLL 或 COM 组件。与其他 .NET Framework 语言不同，Visual C++ 支持互操作性，允许托管代码和非托管代码存在于同一个应用程序中，甚至存在于同一个文件中。然后，可以使用 **/clr** 编译器开关生成 C++ 代码，从而生成托管程序集。最后，在 C# 项目中添加一个对该程序集的引用，并像使用其他托管类那样使用被包装对象。

向 C# 公开 COM 组件

可以使用 C# 项目中的 COM 组件。一般步骤如下所示：

1. 找到要使用的 COM 组件并注册它。使用 regsvr32.exe 注册或注销 COM DLL。
2. 在项目中添加对 COM 组件或类型库的引用。

   添加引用时，Visual Studio 会用到Tlbimp.exe（类型库导入程序），后者将类型库作为输入并输出一个 .NET Framework 互操作程序集。该程序集又称为运行时可调用包装 (RCW)，其中包含了包装类型库中的 COM 类和接口的托管类和接口。Visual Studio 将生成组件的引用添加至项目。

3. 创建在 RCW 中定义的类的实例。而这样会创建 COM 对象的实例。
4. 像使用其他托管对象那样使用该对象。当垃圾回收对该对象进行回收后，COM 对象的实例也会从内存中释放出来。

有关更多信息，请参见 向 .NET Framework 公开 COM 组件。

向 COM 公开 C

COM 客户端可以使用已经正确公开的 C# 类型。公开 C# 类型的基本步骤如下所示：

1. 在 C# 项目中添加互操作特性。

   可以通过修改 Visual C# 项目属性使程序集 COM 可见。有关更多信息，请参见 “程序集信息”对话框。

2. 生成 COM 类型库并对它进行注册以供 COM 使用。

   可以修改 Visual C# 项目属性以自动注册 COM interop 的 C# 组件。Visual Studio 使用 Regasm.exe（程序集注册工具），方法是使用 **/tlb** 命令行切换，其将管理的组件作为输入，以生成类型库。此类型库描述程序集中的 **public** 类型并添加注册表项，以便 COM 客户端可以创建托管类。

有关更多信息，请参见向 COM 公开 .NET Framework 组件和 COM 类示例（C# 编程指南）。

请参阅

Improving Interop Performance

Introduction to COM Interop

Marshaling between Managed and Unmanaged Code

与非托管代码交互操作

Advanced COM Interoperability

C# 编程指南

如何：通过使用 Visual C# 功能访问 Office 互操作对象（C# 编程指南）

Visual C# 2010 引入了可以简化对 Office API 对象的访问的新功能。这些新功能包括命名实参和可选实参、名为 **dynamic** 的新类型，以及在 COM 方法中将实参传递为引用形参（就像它们是值形参）的功能。

在本主题中，你将利用这些新功能来编写创建并显示 Microsoft Office Excel 工作表的代码。然后，你将编写添加包含链接到 Excel 工作表的图标的 Office Word 文档的代码。

若要完成本演练，你的计算机上必须安装 Microsoft Office Excel 2007 和 Microsoft Office Word 2007 或更高版本。

如果你使用的操作系统早于 Windows Vista，请确保安装 .NET Framework 2.0。

> 注意
>
> 以下说明中的某些 Visual Studio 用户界面元素在计算机上出现的名称或位置可能会不同。 这些元素取决于你所使用的 Visual Studio 版本和你所使用的设置。有关详细信息，请参阅个性化 Visual Studio IDE。

创建新的控制台应用程序

1. 启动 Visual Studio。
2. 在“文件”菜单上，指向“新建”，然后单击“项目”。 出现 新建项目 对话框。
3. 在“已安装的模板”窗格中，展开“Visual C#”，然后单击“Windows”。
4. 查看“新建项目”对话框的顶部，确保“.NET Framework 4”（或更高版本）选为目标框架。
5. 在“模板”窗格中，单击“控制台应用程序”。
6. 在“名称”字段中键入项目的名称。
7. 单击“确定”。

   新项目将出现在“解决方案资源管理器”中。

添加引用

1. 在“解决方案资源管理器”中，右键单击你的项目名称，然后单击“添加引用”。将显示“添加引用”对话框。

2. 在“程序集”页上，在“组件名称”列表中选择“Microsoft.Office.Interop.Word”，然后按住 Ctrl 键并选择“Microsoft.Office.Interop.Excel”。如果未看到程序集，你可能需要确保安装并显示它们（参阅如何：安装 Office 主互操作程序集）

3. 单击“确定”。

添加必要的 using 指令

1. 在“解决方案资源管理器”中，右键单击“Program.cs”文件，然后单击“查看代码”。

2. 将以下 **using** 指令添加到代码文件的顶部。

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

创建银行帐户列表

1. 将以下类定义粘贴到“Program.cs”中的 Program 类下。

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

2. 将以下代码添加到 Main 方法，以创建包含两个帐户的 bankAccounts 列表。

```
// Create a list of accounts.
var bankAccounts = new List&lt;Account&gt; {
    new Account {
                  ID = 345678,
                  Balance = 541.27
                },
    new Account {
                  ID = 1230221,
                  Balance = -127.44
                }
};
```

声明将帐户信息导出到 Excel 的方法

1. 将以下方法添加到 Program 类以设置 Excel 工作表。

方法 Add 有一个可选参数，用于指定特定的模板。如果希望使用形参的默认值，你可以借助可选形参（Visual C# 2010 中新增）忽略该形参的实参。由于以下代码中未发送任何参数，**Add** 将使用默认模板并创建新的工作簿。C# 早期版本中的等效语句要求占位符参数：ExcelApp.Workbooks.Add(Type.Missing).

```
static void DisplayInExcel(IEnumerable&lt;Account&gt; accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collectic
    // by property Workbooks. The new workbook becomes the acti
    // Add has an optional parameter for specifying a praticula
    // Because no argument is sent in this example, Add creates
    excelApp.Workbooks.Add();

    // This example uses a single workSheet. The explicit type
    // removed in a later procedure.
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.Acti
}
```

2. 在 DisplayInExcel 的末尾添加以下代码。代码将值插入工作表第一行的前两列。

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

3. 在 DisplayInExcel 的末尾添加以下代码。 **foreach** 循环将帐户列表中的信息放入工作表连续行的前两列。

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

4. 在 DisplayInExcel 的末尾添加以下代码以将列宽调整为适合内容。

```
workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();
```

早期版本的 C# 要求显式强制转换这些操作，因为 ExcelApp.Columns[1] 返回 **Object**，**AutoFit** 为 Excel Range 方法。以下各行显示强制转换。

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

如果程序集由 /link 编译器选项引用或者如果 Excel 的“嵌入互操作类型”属性设置为 true，则 Visual C# 2010 及更高版本会自动将返回的 **Object** 转换为 **dynamic**。True 是此属性的默认值。

运行项目

1. 在 Main 的末尾添加以下行。

```
// Display the list in an Excel spreadsheet.
DisplayInExcel(bankAccounts);
```

2. 按 Ctrl+F5。

   出现包含两个帐户数据的 Excel 工作表。

添加 Word 文档

1. 若要说明 Visual C# 2010 以及更高版本在其他哪些方面增强了 Office 编程，可以使用以下代码打开 Word 应用程序并创建链接到 Excel 工作表的图标。

   将方法 CreateIconInWordDoc（在此步骤后面提供）粘贴到 Program 类中。CreateIconInWordDoc 利用命名参数和可选参数来降低对 Add 和 PasteSpecial 的方法调用的复杂度。这些调用合并了其他两项新功能，这两项新功能在简化对具有引用参数的 COM 方法的调用的 Visual C# 2010 中引入。首先，你可以将实参发送到引用形参，就像它们是值形参一样。即，你可以直接发送值，而无需为每个引用参数创建变量。编译器会生成临时变量以保存参数值，并将在你从调用返回时丢弃变量。其次，你可以忽略参数列表中的 **ref** 关键字。

   Add 方法有四个引用参数，所有引用参数都是可选的。在 Visual C# 2010 或更高版本中，如果希望使用其默认值，可以忽略任何或所有形参的实参。在 Visual C# 2008 以及更早版本中，由于形参是引用形参，因此必须为每个形参提供实参且实参必须是变量。

PasteSpecial 方法可插入剪贴板的内容。该方法有七个引用参数，所有引用参数都是可选的。以下代码为其中两个形参指定实参：Link 用于创建指向剪贴板内容源的连接，DisplayAsIcon 用于将链接显示为图标。在 Visual C# 2010 中，你可以对其中两个形参使用命名实参而忽略其他形参。尽管这些是引用形参，你也不必使用 **ref** 关键字，或者创建变量以实参形式发送。你可以直接发送值。在 Visual C# 2008 以及早期版本中，你必须为每个引用形参发送变量实参。

```
static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of whi
    // optional. Visual C# 2010 allows you to omit arguments fo
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of whic
    // optional. This example uses named arguments to specify v
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or t
    // variables to send in as arguments. You can send the valu
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon:
}
```

在 Visual C# 2008 或早期版本的语言中，需要以下更复杂的代码。

```
static void CreateIconInWordDoc2008()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four parameters, all of which are opt
    // In Visual C# 2008 and earlier versions, an argument has
    // for every parameter. Because the parameters are referenc
    // parameters of type object, you have to create an object
    // for the arguments that represents 'no value'.

    object useDefaultValue = Type.Missing;

    wordApp.Documents.Add(ref useDefaultValue, ref useDefaultVa
        ref useDefaultValue, ref useDefaultValue);

    // PasteSpecial has seven reference parameters, all of whic
    // optional. In this example, only two of the parameters re
    // specified values, but in Visual C# 2008 an argument must
    // for each parameter. Because the parameters are reference
    // you have to contruct variables for the arguments.
    object link = true;
    object displayAsIcon = true;

    wordApp.Selection.PasteSpecial( ref useDefaultValue,
                                    ref link,
                                    ref useDefaultValue,
                                    ref displayAsIcon,
                                    ref useDefaultValue,
                                    ref useDefaultValue,
                                    ref useDefaultValue);
}
```

2. 在 Main 的末尾添加以下语句。

```
// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();
```

3. 在 DisplayInExcel 的末尾添加以下语句。 **Copy** 方法可将工作表添加到剪贴板。

```
// Put the spreadsheet contents on the clipboard. The Copy meth
// optional parameter for specifying a destination. Because no
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
```

4. 按 Ctrl+F5。

   将出现包含图标的 Word 文档。双击该图标以将工作表置于前台。

设置嵌入互操作类型属性

1. 当调用运行时不需要主互操作程序集 (PIA) 的 COM 类型时，可能实现其他增强。删除 PIA 的依赖项可实现版本独立性并且更易于部署。有关不使用 PIA 编程的优势的详细信息，请参阅演练：嵌入托管程序集中的类型（C# 和 Visual Basic）。

   此外，由于可以通过使用类型 **dynamic**（而非 **Object**）表示 COM 方法必需并返回的类型，因此更易于编程。具有类型 **dynamic** 的变量在运行时以前均不会计算，从而消除了显式强制转换的需要。有关详细信息，请参阅使用类型 dynamic（C# 编程指南）。

   在 Visual C# 2010 中，默认行为是嵌入类型信息，而不是使用 PIA。由于该默认行为，因此不需要显式强制转换，之前的几个示例也得到简化。例如，DisplayInExcel 中 worksheet 的声明会写为 Excel._Worksheet workSheet = excelApp.ActiveSheet 而非 Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet。在相同方法中对 **AutoFit** 的调用还将要求在不进行默认行为的情况下显式强制转换，因为 ExcelApp.Columns[1] 返回 **Object**，并且 **AutoFit** 为 Excel 方法。以下代码显示强制转换。

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

2. 若要更改默认行为并使用 PIA 代替嵌入类型信息，请展开“解决方案资源管理器”中的“引用”节点，然后选择“Microsoft.Office.Interop.Excel”或“Microsoft.Office.Interop.Word”。

3. 如果看不到“属性”窗口，请按“F4”。

4. 在属性列表中找到“嵌入互操作类型”，将其值更改为“False”。同样地，你还可以通过在命令提示符下使用 /reference 编译器选项代替 /link 进行编译。

将其他格式添加到表格

1. 将在 DisplayInExcel 中对 **AutoFit** 的两个调用替换为以下语句。

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

AutoFormat 方法有七个值参数，每个值参数都是可选的。使用命名参数和可选参数，你可以为这些参数中的所有或部分提供参数，也可以不为它们中的任何一个提供。在上一条语句中，仅为其中一个形参 Format 提供实参。由于 Format 是参数列表中的第一个参数，因此无需提供参数名称。但是，如果包含参数名称，语句则可能更易于理解，如以下代码所示。

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

2. 按 Ctrl+F5 查看结果。其他格式在 XlRangeAutoFormat 枚举中列出。
3. 将步骤 1 中的语句与以下代码比较（以下代码显示 Visual C# 2008 或早期版本中要求的参数）。

```
// The AutoFormat method has seven optional value parameters. T
// following call specifies a value for the first parameter, ar
// the default values for the other six.

// Call to AutoFormat in Visual C# 2008\. This code is not part
// current solution.
excelApp.get_Range("A1", "B4").AutoFormat(Excel.XlRangeAutoForm
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Typ
    Type.Missing);
```

以下代码显示完整示例。

```
using System;
using System.Collections.Generic;
using System.Linq;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgramminWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
```

```
                new Account {
                              ID = 345678,
                              Balance = 541.27
                            },
                new Account {
                              ID = 1230221,
                              Balance = -127.44
                            }
            };

            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that li
            // the spreadsheet.
            CreateIconInWordDoc();
        }

        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the colle
            // by property Workbooks. The new workbook becomes the 
            // Add has an optional parameter for specifying a prati
            // Because no argument is sent in this example, Add cre
            excelApp.Workbooks.Add();

            // This example uses a single workSheet.
            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            // Earlier versions of C# require explicit casting.
            //Excel._Worksheet workSheet = (Excel.Worksheet)excelAp

            // Establish column headings in cells A1 and B1.
            workSheet.Cells[1, "A"] = "ID Number";
            workSheet.Cells[1, "B"] = "Current Balance";

            var row = 1;
            foreach (var acct in accounts)
            {
                row++;
                workSheet.Cells[row, "A"] = acct.ID;
                workSheet.Cells[row, "B"] = acct.Balance;
            }

            workSheet.Columns[1].AutoFit();
            workSheet.Columns[2].AutoFit();

            // Call to AutoFormat in Visual C# 2010\. This statemen
            // two calls to AutoFit.
```

```
            workSheet.Range["A1", "B3"].AutoFormat(
                Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

            // Put the spreadsheet contents on the clipboard. The C
            // optional parameter for specifying a destination. Bec
            // is sent, the destination is the Clipboard.
            workSheet.Range["A1:B3"].Copy();
        }

        static void CreateIconInWordDoc()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;

            // The Add method has four reference parameters, all of
            // optional. Visual C# 2010 allows you to omit argument
            // the default values are what you want.
            wordApp.Documents.Add();

            // PasteSpecial has seven reference parameters, all of
            // optional. This example uses named arguments to speci
            // for two of the parameters. Although these are refere
            // parameters, you do not need to use the ref keyword,
            // variables to send in as arguments. You can send the
            wordApp.Selection.PasteSpecial(Link: true, DisplayAsIco
        }
    }

    public class Account
    {
        public int ID { get; set; }
        public double Balance { get; set; }
    }
}
```

请参阅

Type.Missing

dynamic（C# 参考）

使用类型 dynamic（C# 编程指南）

命名实参和可选实参（C# 编程指南）

如何：在 Office 编程中使用命名参数和可选参数（C# 编程指南）

如何：在 COM 互操作编程中使用索引属性（C# 编程指南）

索引属性改进了在 C# 编程中使用具有参数的 COM 属性的方式。将索引属性与 Visual C# 2010 中引入的其他功能（如命名参数和可选参数、新的 dynamic 类型以及嵌入的类型信息）一起使用，可增强 Microsoft Office 编程功能。

在早期版本的 C# 中，只有在 **get** 方法不具有任何参数且 **set** 方法有且仅有一个值参数时，才能将方法作为属性进行访问。但是，并非所有 COM 属性都符合上述限制。例如，Excel 的 Range 属性具有一个 **get** 访问器，该访问器需要一个表示范围名称的参数。以前，由于您无法直接访问 **Range** 属性，因此您必须改用 **get_Range** 方法，如以下示例所示。

```
// Visual C# 2008 and earlier.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

利用索引属性，您可以改为编写以下代码：

```
// Visual C# 2010.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.Range["A1"];
```

注意

上面的示例还使用可选参数功能（在 Visual C# 2010 中引入），此功能可使您忽略 **Type.Missing**。

与此类似，若要在 Visual C# 2008 及更早版本中设置 Range 对象的 **Value** 属性的值，需要两个参数。一个参数为用于指定范围值的类型的可选形参提供实参。另一个参数提供 **Value** 属性的值。在 Visual C# 2010 之前的版本中，C# 仅允许一个参数。因此，您必须使用 **set_Value** 方法，或者使用另一个属性 Value2，而不能使用常规的 set 方法。下面的示例演示了这些技术。这两种技术都将 A1 单元格的值设置为 Name。

```
// Visual C# 2008.
targetRange.set_Value(Type.Missing, "Name");
// Or
targetRange.Value2 = "Name";
```

利用索引属性，您可以改为编写以下代码。

```
// Visual C# 2010.
targetRange.Value = "Name";
```

您不能创建自己的索引属性。该功能只支持使用现有的索引属性。

下面的代码显示一个完整的示例。有关如何设置访问 Office API 的项目的更多信息，请参见如何：通过使用 Visual C# 功能访问 Office 互操作对象（C# 编程指南）。

```
// You must add a reference to Microsoft.Office.Interop.Excel to ru
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
            //CSharp2008();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add();
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.Range["A1"];
            targetRange.Value = "Name";
        }

        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.get_Range("A1", Type
            targetRange.set_Value(Type.Missing, "Name");
            // Or
            //targetRange.Value2 = "Name";

        }
    }
}
```

请参阅

命名实参和可选实参（C# 编程指南）

dynamic（C# 参考）

使用类型 dynamic（C# 编程指南）

如何：在 Office 编程中使用命名参数和可选参数（C# 编程指南）

如何：通过使用 Visual C# 功能访问 Office 互操作对象（C# 编程指南）

演练：Office 编程（C# 和 Visual Basic）

如何：使用平台调用播放波形文件（C# 编程指南）

下面的 C# 代码示例演示如何在 Windows 操作系统上使用平台调用服务来播放波形声音文件。

此代码示例使用 **DllImport** 将 **winmm.dll** 的 **PlaySound** 方法入口点作为 Form1 PlaySound() 导入。该示例具有一个带有一个按钮的简单 Windows 窗体。单击该按钮可打开标准 Windows OpenFileDialog 对话框，以便打开要播放的文件。选择波形文件后，将使用 winmm.DLL 程序集方法的 **PlaySound()** 方法来播放此文件。有关 winmm.dll 的 **PlaySound** 方法的更多信息，请参见 Using the PlaySound function with Waveform-Audio Files（使用 PlaySound 函数播放波形音频文件）。浏览并选择一个带 .wav 扩展名的文件，然后单击“打开”以使用平台调用播放该波形文件。将在文本框中显示选择的文件的完整路径。

“打开文件”对话框将使用以下筛选器设置只显示具有 .wav 扩展名的文件：

```
dialog1.Filter = "Wav Files (*.wav)|*.wav";
```

```
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace WinSound
{
    public partial class Form1 : Form
    {
        private TextBox textBox1;
        private Button button1;

        public Form1()  //constructor
        {
            InitializeComponent();
        }

        [System.Runtime.InteropServices.DllImport("winmm.DLL", Entr
        private static extern bool PlaySound(string szSound, Syster

        [System.Flags]
        public enum PlaySoundFlags : int
        {
            SND_SYNC = 0x0000,
            SND_ASYNC = 0x0001,
            SND_NODEFAULT = 0x0002,
            SND_LOOP = 0x0008,
            SND_NOSTOP = 0x0010,
            SND_NOWAIT = 0x00002000,
            SND_FILENAME = 0x00020000,
            SND_RESOURCE = 0x00040004
        }

        private void button1_Click (object sender, System.EventArgs
        {
            OpenFileDialog dialog1 = new OpenFileDialog();

            dialog1.Title = "Browse to find sound file to play";
            dialog1.InitialDirectory = @"c:\";
            dialog1.Filter = "Wav Files (*.wav)|*.wav";
            dialog1.FilterIndex = 2;
            dialog1.RestoreDirectory = true;

            if(dialog1.ShowDialog() == DialogResult.OK)
            {
                textBox1.Text = dialog1.FileName;
                PlaySound (dialog1.FileName, new System.IntPtr(), F
            }
        }
    }
}
```

编译代码

编译代码

1. 在 Visual Studio 中创建一个新的 C# Windows 应用程序项目，并命名为 WinSound。

2. 复制上面的代码并将其粘贴到 Form1.cs 文件中以覆盖原来的内容。

3. 复制下面的代码，并将它粘贴到 Form1.Designer.cs 文件的 InitializeComponent() 方法中现有代码的后面。

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Clic
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "FIle path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

4. 编译并运行代码。

.NET Framework 安全性

有关更多信息，请参见 .NET Framework Security（.NET Framework 安全性）。

请参阅

C# 编程指南

互操作性概述（C# 编程指南）

互操作性概述（C# 编程指南）

A Closer Look at Platform Invoke

用平台调用封送数据

演练：Office 编程（C# 和 Visual Basic）

Visual Studio 2010 在 C# 和 Visual Basic 中引入了改进 Microsoft Office 编程的新功能。每种语言都增加了其他语言中已经存在的功能。

C# 中的新功能包括命名参数和可选参数、具有 **dynamic** 类型的返回值，以及在 COM 编程中忽略 **ref** 关键字和访问索引属性的功能。Visual Basic 中的新功能包括自动实现的属性、Lambda 表达式语句和集合初始值设定项。

两种语言都支持嵌入类型信息，从而允许在不向用户的计算机部署主互操作程序集 (PIA) 的情况下部署与 COM 组件交互的程序集。有关详细信息，请参阅演练：嵌入托管程序集中的类型（C# 和 Visual Basic）。

本演练演示 Office 编程环境中的新功能，但这些新功能在常规编程中也极为有用。在本演练中，你将首先使用 Excel 外接应用程序来创建 Excel 工作簿。然后，你将创建包含工作簿链接的 Word 文档。最后，你将看到可以如何开启和关闭 PIA 依赖项。

系统必备

若要完成本演练，你的计算机上必须安装有 Microsoft Office Excel 2013（或者 2007 或更高版本）和 Microsoft Office Word 2013（或者 2007 或更高版本）。

如果你使用的操作系统早于 Windows Vista，请确保安装 .NET Framework 2.0。

> 注意
>
> 以下说明中的某些 Visual Studio 用户界面元素在计算机上出现的名称或位置可能会不同。这些元素取决于你所使用的 Visual Studio 版本和你所使用的设置。有关详细信息，请参阅个性化 Visual Studio IDE。

设置 Excel 外接应用程序

1. 启动 Visual Studio。
2. 在“文件”菜单上，指向“新建”，然后单击“项目”。
3. 在“安装的模板”窗格中，展开“Visual Basic”或“Visual C#”，再展开“Office”，然后单击“2013”（或“2010”或“2007”）。
4. 在“模板”窗格中，单击“Excel 2013 外接应用程序”（或“Excel 2010 外接应用程序”或“Excel 2007 外接应用程序”）。
5. 查看“模板”窗格的顶部，确保“.NET Framework 4”或更高版本出现在“目标框架”框中。

6. 如果需要，在“名称”框中键入项目的名称。
7. 单击“确定”。
8. 新项目将出现在“解决方案资源管理器”中。

添加引用

1. 在“解决方案资源管理器”中，右键单击你的项目名称，然后单击“添加引用”。将显示“添加引用”对话框。
2. 在“程序集”选项卡上，在“组件名称”列表中选择“Microsoft.Office.Interop.Excel”版本 15.0.0.0（针对 Excel 2010，选择版本 14.0.0.0；针对 Excel 2007，选择版本 12.0.0.0），然后按住 Ctrl 键并选择“Microsoft.Office.Interop.Word”版本 15.0.0.0（针对 Word 2010，选择版本 14.0.0.0；针对 Word 2007，选择版本 12.0.0.0）。如果未看到程序集，你可能需要确保安装并显示它们（参阅如何：安装 Office 主互操作程序集）。
3. 单击“确定”。

添加必要的 Imports 语句或 using 指令

1. 在“解决方案资源管理器”中，右键单击“ThisAddIn.vb”或“ThisAddIn.cs”文件，然后单击“查看代码”。
2. 将以下 **Imports** 语句 (Visual Basic) 或 **using** 指令 (C#) 添加到代码文件的顶部（如果不存在）。

   VB

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

创建银行帐户列表

1. 在“解决方案资源管理器”中，右键单击你的项目名称，单击“添加”，然后单击“类”。如果使用的是 Visual Basic，则将类命名为 Account.vb；如果使用的是 C#，则将类命名为 Account.cs。单击“添加”。
2. 将 Account 类的定义替换为以下代码。类定义使用“自动实现的属性”，在 Visual Studio 2010 中，是 Visual Basic 的新功能。有关详细信息，请参阅自动实现的属性 (Visual Basic)。

   VB

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

3. 若要创建包含两个帐户的 bankAccounts 列表，请将以下代码添加到 ThisAddIn.vb 或 ThisAddIn.cs 中的 ThisAddIn*Startup* 方法。列表声明使用“集合初始值设定项”_，在 Visual Studio 2010 中，是 Visual Basic 的新功能。有关详细信息，请参阅集合初始值设定项 (Visual Basic)。

   VB

```
var bankAccounts = new List&lt;Account&gt;
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

将数据导出到 Excel

1. 在相同的文件中，将以下方法添加到 ThisAddIn 类。该方法设置 Excel 工作薄并将数据导出到工作簿。

   VB

```
void DisplayInExcel(IEnumerable&lt;Account&gt; accounts,
           Action&lt;Account, Excel.Range&gt; DisplayFunc)
{
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
    excelApp.Range["A2"].Select();

    foreach (var ac in accounts)
    {
        DisplayFunc(ac, excelApp.ActiveCell);
        excelApp.ActiveCell.Offset[1, 0].Select();
    }
    // Copy the results to the Clipboard.
    excelApp.Range["A1:B3"].Copy();
}
```

此方法使用 C# 的两项新功能。Visual Basic 中已存在这两项功能。

- 方法 Add 有一个“可选参数”，用于指定特定的模板。如果希望使用形参的默认值，你可以借助可选形参（Visual C# 2010 中新增）忽略该形参的实参。由于上一个示例中未发送任何参数，**Add** 将使用默认模板并创建新的工作簿。C# 早期版本中的等效语句要求占位符参数：excelApp.Workbooks.Add(Type.Missing).

  有关详细信息，请参阅命名实参和可选实参（C# 编程指南）。

- Range 对象的 **Range** 和 **Offset** 属性使用“索引属性”功能。此功能允许你通过以下典型 C# 语法从 COM 类型使用这些属性。索引属性还允许你使用 **Range** 对象的 **Value** 属性，因此不必使用 **Value2** 属性。 **Value** 属性已编入索引，但索引是可选的。在以下示例中，可选参数和索引属性配合使用。

  ```
  // Visual C# 2010 provides indexed properties for COM progr
  excelApp.Range["A1"].Value = "ID";
  excelApp.ActiveCell.Offset[1, 0].Select();
  ```

  在早期版本的语言中，需要以下特殊语法。

  ```
  // In Visual C# 2008, you cannot access the Range, Offset,
  // properties directly.
  excelApp.get_Range("A1").Value2 = "ID";
  excelApp.ActiveCell.get_Offset(1, 0).Select();
  ```

你不能创建自己的索引属性。该功能仅支持使用现有索引属性。

有关详细信息，请参阅如何：在 COM 互操作编程中使用索引属性（C# 编程指南）。

2. 在 DisplayInExcel 的末尾添加以下代码以将列宽调整为适合内容。

VB

```
excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

这些添加展示 C# 2010 中的另一项新功能：处理从 COM 主机返回的 **Object** 值（如 Office），就像它们具有 dynamic 类型一样。当“嵌入互操作类型”设置为其默认值 **True** 时，或者由 /link 编译器选项引用程序集时，自动发生这种情况。键入 **dynamic** 允许后期绑定（Visual Basic 已提供该功能）并可避免 Visual C# 2008 和早期版本的语言中要求的显式强制转换。

例如，excelApp.Columns[1] 返回 **Object**，并且 **AutoFit** 是 Excel 的 Range 方法。如果没有 **dynamic**，你必须将 excelApp.Columns[1] 返回的对象强制转换为 Range 的实例，然后才能调用 AutoFit 方法。

```
// Casting is required in Visual C# 2008.
((Excel.Range)excelApp.Columns[1]).AutoFit();

// Casting is not required in Visual C# 2010.
excelApp.Columns[1].AutoFit();
```

有关嵌入互操作类型的详细信息，请参阅本主题后面部分的“查找 PIA 引用”和“还原 PIA 依赖项”程序。有关 **dynamic** 的详细信息，请参阅 dynamic（C# 参考）或 使用类型 dynamic（C# 编程指南）。

调用 DisplayInExcel

1. 在 ThisAddIn_StartUp 方法的末尾添加以下代码。对 DisplayInExcel 的调用包含两个参数。第一个参数是要处理的帐户列表的名称。第二个参数是定义如何处理数据的多行 lambda 表达式。每个帐户的 ID 和 balance 值都显示在相邻的单元格中，如果余额小于零，则相应的行显示为红色。多行 lambda 表达式是 Visual Basic 2010 中的新功能。有关详细信息，请参阅Lambda 表达式 (Visual Basic)。

VB

```
DisplayInExcel(bankAccounts, (account, cell) =&gt;
// This multiline lambda expression sets custom processing rule
// for the bankAccounts.
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance &lt; 0)
    {
        cell.Interior.Color = 255;
        cell.Offset[0, 1].Interior.Color = 255;
    }
});
```

2. 若要运行程序，请按 F5。出现包含帐户数据的 Excel 工作表。

添加 Word 文档

1. 在 ThisAddIn_StartUp 方法末尾添加以下代码，以创建包含指向 Excel 工作簿的链接的 Word 文档。

   VB

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true)
```

   此代码展示 C# 中的几项新功能：在 COM 编程中忽略 **ref** 的功能、命名参数以及可选参数。Visual Basic 中已存在这些功能。 PasteSpecial 方法有七个参数，每个参数都是可选引用参数。在 Visual C# 2010 之前，你必须为这七个形参定义用作实参的对象变量，即使你没有有意义的值发送。通过命名实参和可选实参，你可以指定希望按名称访问的形参并仅将实参发送到这些形参。在本示例中，发送实参以指示应创建指向剪贴板上工作簿的链接（形参 Link）并指示该链接应在 Word 文档中显示为图标（形参 DisplayAsIcon）。Visual C# 2010 还允许你忽略这些参数的 **ref** 关键字。将 Visual C# 2008 的以下代码段与 Visual C# 2010 中需要的单行进行比较：

```
// Call to PasteSpecial in Visual C# 2008.
object iconIndex = Type.Missing;
object link = true;
object placement = Type.Missing;
object displayAsIcon = true;
object dataType = Type.Missing;
object iconFileName = Type.Missing;
object iconLabel = Type.Missing;
wordApp.Selection.PasteSpecial(ref iconIndex,
                               ref link,
                               ref placement,
                               ref displayAsIcon,
                               ref dataType,
                               ref iconFileName,
                               ref iconLabel);

// Call to PasteSpecial in Visual C# 2010.
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true)
```

运行应用程序

1. 按 F5 运行该应用程序。Excel 启动并显示包含 bankAccounts 中两个帐户的信息的表。然后，出现包含指向 Excel 表的 Word 文档。

清理已完成的项目

1. 在 Visual Studio 中，单击“生成”菜单上的“清理解决方案”。否则，每次在计算机上打开 Excel 时都会运行外接应用程序。

查找 PIA 引用

1. 再次运行应用程序，但不单击“清理解决方案”。
2. 在“开始”菜单上，单击“所有程序”。接下来依次单击“Microsoft Visual Studio 2013”、“Visual Studio 工具”、“Visual Studio 命令提示符 (2013)”。
3. 在 Visual Studio 命令提示符 (2013) 窗口中键入 **ildasm**，然后按 Enter。此时将出现 IL DASM 窗口。
4. 在 IL DASM 窗口的“文件”菜单上，单击“打开”。双击“Visual Studio 2013”，然后双击“项目”。打开项目的文件夹，在 bin/Debug 文件夹中查找 项目名称.dll。双击 项目名称.dll。新窗口将显示项目的属性以及对其他模块和程序集的引用。注意，命名空间 **Microsoft.Office.Interop.Excel** 和 **Microsoft.Office.Interop.Word** 包含在程序集中。在 Visual Studio 2013 中，编译器默认将你需要的类型从引用的 PIA 导入程序集。

有关详细信息，请参阅如何：查看程序集内容。

5. 双击“清单”图标。此时将出现包含程序集列表的窗口，这些程序集包含项目所引用的项。 **Microsoft.Office.Interop.Excel** 和 **Microsoft.Office.Interop.Word** 未包含在列表中。由于项目需要的类型已导入程序集中，因此不需要引用 PIA。这使得部署变得更加容易。用户的计算机上不必存在 PIA，因为应用程序不需要部署特定版本的 PIA，应用程序可设计为与多个版本的 Office 配合使用，前提是所有版本中都存在必要的 API。

   由于不再需要部署 PIA，你可以提前创建可与多个版本的 Office（包括之前的版本）配合使用的应用程序。但是，仅当你的代码不使用你当前所使用 Office 版本中不可用的任何 API 时，此情况才适用。特殊 API 在早期版本中是否可用并不始终明确，因此不建议使用早期版本的 Office。

6. 关闭清单窗口和程序集窗口。

注意
在 Office 2003 以前，Office 并不发布 PIA。因此，生成适用于 Office 2002 或早期版本的互操作程序集的唯一方法是导入 COM 引用。

还原 PIA 依赖项

1. 在“解决方案资源管理器”中，单击“显示所有文件”按钮。展开“引用”文件夹并选择“Microsoft.Office.Interop.Excel”。按 F4 以显示“属性”窗口。
2. 在“属性”窗口中，将“嵌入互操作类型”属性从“True”更改为“False”。
3. 对 **Microsoft.Office.Interop.Word** 重复此程序中的步骤 1 和 2。
4. 在 C# 中，在 DisplayInExcel 方法的末尾注释掉对 Autofit 的两次调用。
5. 按 F5 以验证项目是否仍正确运行。
6. 重复上一个程序的步骤 1-3 以打开程序集窗口。注意，**Microsoft.Office.Interop.Word** 和 **Microsoft.Office.Interop.Excel** 不再位于嵌入程序集列表中。
7. 双击“清单”图标并滚动引用程序集的列表。 **Microsoft.Office.Interop.Word** 和 **Microsoft.Office.Interop.Excel** 均位于列表中。由于应用程序引用 Excel 和 Word PIA 并且“嵌入互操作类型”属性设置为“False”，因此最终用户的计算机上必须存在两个程序集。
8. 在 Visual Studio 中，单击“生成”菜单上的“清理解决方案”以清理完成的项目。

请参阅

自动实现的属性 (Visual Basic)

自动实现的属性（C# 编程指南）

集合初始值设定项 (Visual Basic)

对象和集合初始值设定项（C# 编程指南）

可选参数 (Visual Basic)

按位置和名称传递参数 (Visual Basic)

命名实参和可选实参（C# 编程指南）

早期绑定和后期绑定 (Visual Basic)

dynamic（C# 参考）

使用类型 dynamic（C# 编程指南）

Lambda 表达式 (Visual Basic)

Lambda 表达式（C# 编程指南）

如何：在 COM 互操作编程中使用索引属性（C# 编程指南）

演练：嵌入 Microsoft Office 程序集中的类型信息（C# 和 Visual Basic）

演练：嵌入托管程序集中的类型（C# 和 Visual Basic）

演练：创建你的第一个 Excel VSTO 外接程序

COM 互操作 (Visual Basic)

互操作性（C# 编程指南）

COM 类示例（C# 编程指南）

下面是一个公开为 COM 对象的类的示例。将这些代码放置到 .cs 文件中并添加到您的项目中后，请将**“Register for COM Interop”**属性设置为**“True”**。有关更多信息，请参见NIB: How to: Register a Component for COM Interop。

向 COM 公开 Visual C# 对象要求声明一个类接口、一个事件接口（如果需要）和类本身。类成员必须遵循下列规则才能对 COM 可见：

- 类必须是公共的。
- 属性、方法和事件必须是公共的。
- 属性和方法必须在类接口上声明。
- 事件必须在事件接口中声明。

其他没有在这些接口中声明的类的公共成员对于 COM 是不可见的，但它们对于其他 .NET Framework 对象将是可见的。

若要向 COM 公开属性和方法，必须在类接口上声明这些属性和方法，并用 **DispId** 特性予以标记，然后在类中实现它们。成员在接口中声明的顺序即是用于 COM vtable 的顺序。

若要从类中公开事件，必须在事件接口上声明这些事件，并用 **DispId** 特性予以标记。该类不应实现此接口。

类实现类接口；它可以实现多个接口，但第一个实现将作为默认类接口。在此处实现向 COM 公开的方法和属性。它们必须标记为是公共的，并且必须与类接口中的声明匹配。同时，在此处声明由类引发的事件。它们必须标记为是公共的，并且必须与事件接口中的声明匹配。

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
    public interface ComClass1Interface
    {
    }

    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
        InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ComClass1Events
    {
    }

    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
        ClassInterface(ClassInterfaceType.None),
        ComSourceInterfaces(typeof(ComClass1Events))]
    public class ComClass1 : ComClass1Interface
    {
    }
}
```

请参阅

C# 编程指南

互操作性（C# 编程指南）

“项目设计器”->“生成”页 (C#)

LINQ 查询表达式（C# 编程指南）

语言集成查询 (LINQ) 是一组技术的名称，这些技术建立在将查询功能直接集成到 C# 语言（以及 Visual Basic 和可能的任何其他 .NET 语言）的基础上。借助于 LINQ，查询现在已是高级语言构造，就如同类、方法、事件等等。

对于编写查询的开发人员来说，LINQ 最明显的“语言集成”部分是查询表达式。查询表达式是使用 C# 3.0 中引入的声明性查询语法编写的。通过使用查询语法，您甚至可以使用最少的代码对数据源执行复杂的筛选、排序和分组操作。您使用相同的基本查询表达式模式来查询和转换 SQL 数据库、ADO.NET 数据集、XML 文档和流以及 .NET 集合中的数据。

下面的示例演示了完整的查询操作。完整操作包括创建数据源、定义查询表达式，以及在 **foreach** 语句中执行查询。

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

有关 C# 中的 LINQ 基础知识的更多信息，请参见 Getting Started with LINQ in C#。

查询表达式概述

- 查询表达式可用于查询和转换来自任意支持 LINQ 的数据源中的数据。例如，单个查询可以从 SQL 数据库检索数据，并生成 XML 流作为输出。

- 查询表达式容易掌握，因为它们使用许多常见的 C# 语言构造。有关更多信息，请参见 Getting Started with LINQ in C#。
- 查询表达式中的变量都是强类型的，但许多情况下您不需要显式提供类型，因为编译器可以推断类型。有关更多信息，请参见Type Relationships in LINQ Query Operations (C#)。
- 在您循环访问 **foreach** 语句中的查询变量之前，不会执行查询。有关更多信息，请参见 Introduction to LINQ Queries (C#)。
- 在编译时，根据 C# 规范中设置的规则将查询表达式转换为“标准查询运算符”方法调用。任何可以使用查询语法表示的查询也可以使用方法语法表示。但是，在大多数情况下，查询语法更易读和简洁。有关更多信息，请参见 C# 语言规范和Standard Query Operators Overview。
- 作为编写 LINQ 查询的一项规则，建议尽量使用查询语法，只在必需的情况下才使用方法语法。这两种不同形式在语义或性能上没有区别。查询表达式通常比用方法语法编写的等效表达式更易读。
- 一些查询操作，如 Count<TSource> 或 Max，没有等效的查询表达式子句，因此必须表示为方法调用。方法语法可以通过多种方式与查询语法组合。有关更多信息，请参见Query Syntax and Method Syntax in LINQ (C#)。
- 查询表达式可以编译为表达式树或委托，具体取决于查询所应用到的类型。IEnumerable<T> 查询编译为委托。 IQueryable 和 IQueryable<T> 查询编译为表达式树。有关更多信息，请参见表达式树（C# 和 Visual Basic）。

下表列出了一些主题，提供有关常规任务的查询和代码示例的其他信息。

主题	说明
查询表达式基础（C# 编程指南）	介绍基本查询概念并提供 C# 查询语法的示例。
如何：在 C# 中编写 LINQ 查询	提供若干基本查询表达式类型的示例。
如何：在查询表达式中处理异常（C# 编程指南）	如何以及何时将可能会引发异常的代码移出查询表达式。
How to: Populate Object Collections from Multiple Sources (LINQ)	如何使用 **select** 语句将来自不同源的数据合并为新类型。
如何：对查询结果进行分组（C# 编程指南）	演示使用 **group** 子句的不同方法。
如何：创建嵌套组（C# 编程指南）	演示如何创建嵌套组。
如何：对分组操作执行子查询（C# 编程指南）	演示如何使用查询中的子表达式作为新查询的数据源。
	演示如何实现线程安全的标准查询运算

（C# 编程指南）	符，该运算符可对流式数据源执行分组操作。
如何：在运行时动态指定谓词筛选器（C# 编程指南）	演示如何为 **where** 子句的相等比较提供任意数目的值。
如何：在内存中存储查询结果（C# 编程指南）	阐释如何具体化和存储查询结果，而不必使用 **foreach** 循环。
如何：从方法中返回查询（C# 编程指南）	演示如何从方法返回查询变量，以及如何将它们作为输入参数传递给方法。
如何：执行自定义联接操作（C# 编程指南）	演示如何基于任何类型的谓词函数执行联接运算。
如何：使用复合键进行联接（C# 编程指南）	演示如何基于多个匹配键联接两个源。
如何：对 Join 子句的结果进行排序（C# 编程指南）	演示如何对联接运算生成的序列进行排序。
如何：执行内部联接（C# 编程指南）	演示如何在 LINQ 中执行内联。
如何：执行分组联接（C# 编程指南）	演示如何在 LINQ 中生成已分组的联接。
如何：执行左外部联接（C# 编程指南）	演示如何在 LINQ 中生成左外部联接。
如何：在查询表达式中处理 Null 值（C# 编程指南）	演示如何在 LINQ 查询中处理 null 值。

请参阅

C# 编程指南

LINQ (Language-Integrated Query)

Walkthrough: Writing Queries in C# (LINQ)

Basic LINQ Query Operations (C#)

Query Syntax and Method Syntax in LINQ (C#)

Standard Query Operators Overview

查询关键字（C# 参考）

How Linq to Objects Queries Work（Linq to Objects 查询如何工作）

Reading and Writing Queries（读取和编写查询）

What is a collection?（什么是集合？）

Link to Everything: A List of LINQ Providers（所有链接：LINQ 提供程序列表）

查询表达式基础（C# 编程指南）

什么是查询？它有什么用途？

“查询”是指一组指令，这些指令描述要从一个或多个给定数据源检索的数据以及返回的数据应该使用的格式和组织形式。查询不同于它所产生的结果。

通常，源数据会在逻辑上组织为相同种类的元素序列。SQL 数据库表包含一个行序列。与此类似，ADO.NET DataTable 包含一个 DataRow 对象序列。在 XML 文件中，有一个 XML 元素“序列”（不过这些元素按分层形式组织为树结构）。内存中的集合包含一个对象序列。

从应用程序的角度来看，原始源数据的具体类型和结构并不重要。应用程序始终将源数据视为一个 IEnumerable<T> 或 IQueryable<T> 集合。在 LINQ to XML 中，源数据显示为一个 **IEnumerable**<XElement>。在 LINQ to DataSet 中，它是一个 **IEnumerable**<DataRow>。在 LINQ to SQL 中，它是您定义用来表示 SQL 表中数据的任何自定义对象的 **IEnumerable** 或 **IQueryable**。

指定此源序列后，查询可以进行下列三项工作之一：

- 检索一个元素子集以产生一个新序列，但不修改单个元素。然后，查询可以按各种方式对返回的序列进行排序或分组，如下面的示例所示（假定 scores 是 **int[]**）：

```
IEnumerable&lt;int&gt; highScoresQuery =
    from score in scores
    where score &gt; 80
    orderby score descending
    select score;
```

- 如上一个示例所述检索一个元素序列，但是将这些元素转换为具有新类型的对象。例如，查询可以只从数据源中的某些客户记录检索姓氏。或者，查询可以检索完整的记录，再使用它构建另一个内存中对象类型甚至 XML 数据，然后生成最终的结果序列。下面的示例演示了从 **int** 到 **string** 的转换。请注意 highScoresQuery 的新类型。

```
IEnumerable&lt;string&gt; highScoresQuery2 =
    from score in scores
    where score &gt; 80
    orderby score descending
    select String.Format("The score is {0}", score);
```

- 检索有关源数据的单一值，例如：
  - 符合某个条件的元素的数量。

- 具有最大值或最小值的元素。
- 符合某个条件的第一个元素，或一组指定元素中的特定值之和。例如，下面的查询从 scores 整数数组中返回高于 80 的分数的数量。

```
int highScoreCount =
    (from score in scores
     where score &gt; 80
     select score)
     .Count();
```

在上一个示例中，请注意在 **Count** 方法调用之前的查询表达式两旁使用了括号。另一种表示方式是使用一个新变量来存储具体结果。此技术的可读性更好，因为它将存储查询的变量与存储结果的查询区分开来。

```
IEnumerable&lt;int&gt; highScoresQuery3 =
    from score in scores
    where score &gt; 80
    select score;

int scoreCount = highScoresQuery3.Count();
```

在上一个示例中，查询是在 **Count** 调用中执行的，因为 **Count** 必须循环访问结果以便确定 highScoresQuery 返回的元素数量。

什么是查询表达式？

“查询表达式”是用查询语法表示的查询，是一流的语言构造。它就像任何其他表达式一样，并且可以用在 C# 表达式有效的任何上下文中。查询表达式由一组用类似于 SQL 或 XQuery 的声明性语法编写的子句组成。每个子句又包含一个或多个 C# 表达式，而这些表达式本身又可能是查询表达式或包含查询表达式。

查询表达式必须以 from 子句开头，并且必须以 select 或 group 子句结尾。在第一个 **from** 子句和最后一个 **select** 或 **group** 子句之间，查询表达式可以包含一个或多个下列可选子句：where、orderby、join、let 甚至附加的 from 子句。还可以使用 into 关键字使 **join** 或 **group** 子句的结果能够充当同一查询表达式中附加查询子句的源。

查询变量

在 LINQ 中，查询变量是任何存储查询（而非查询结果）的变量。更具体地说，查询变量始终是一个可枚举的类型，当在 **foreach** 语句中或在对其 **IEnumerator.MoveNext** 方法的直接调用中循环访问它时，它会生成一序列元素。

下面的代码示例演示了一个简单的查询表达式，它含有一个数据源、一个筛选子句和一个排序子句，但不对源元素进行转换。 **select** 子句结束了该查询。

```
static void Main()
{
    // Data source.
    int[] scores = { 90, 71, 82, 93, 75, 82 };

    // Query Expression.
    IEnumerable<int> scoreQuery = //query variable
        from score in scores //required
        where score > 80 // optional
        orderby score descending // optional
        select score; //must end with select or group

    // Execute the query to produce the results
    foreach (int testScore in scoreQuery)
    {
        Console.WriteLine(testScore);
    }
}
// Outputs: 93 90 82 82
```

在上一个示例中，scoreQuery 是一个查询变量，有时简称为“查询”。查询变量并不存储实际的结果数据（这些数据是在 **foreach** 循环中产生的）。另外，当 **foreach** 语句执行时，查询结果并不是通过查询变量 scoreQuery 返回的。相反，它们是通过迭代变量 testScore 返回的。可以在另一个 **foreach** 循环中迭代 scoreQuery 变量。只要该变量和数据源都没有修改，该变量都将产生相同的结果。

查询变量可以存储用查询语法或方法语法（或二者的组合）表示的查询。在下面的示例中，queryMajorCities 和 queryMajorCities2 都是查询变量：

```
//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Populatio
```

另一方面，下面的两个示例演示了不是查询变量的变量，即使每个变量都用查询进行了初始化。它们不是查询变量的原因是它们存储了结果：

```
int highestScore =
    (from score in scores
     select score)
    .Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

int highScore = scoreQuery.Max();

List<City> largeCitiesList =
    (from country in countries
     from city in country.Cities
     where city.Population > 10000
     select city)
       .ToList();

// or split the expression
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;

List<City> largeCitiesList2 = largeCitiesQuery.ToList();
```

注意

在 LINQ 文档中，存储查询的变量在其名称中包含单词“query”，而存储实际结果的变量在其名称中不包含单词“query”。

有关用于表示查询的不同方式的更多信息，请参见Query Syntax and Method Syntax in LINQ (C#)。

查询变量的显式类型化和隐式类型化

本文档通常提供查询变量的显式类型，以便演示查询变量和 select 子句之间的类型关系。但是，也可以使用 var 关键字指示编译器在编译时推断查询变量（或任何其他本地变量）的类型。例如，还可以使用隐式类型化表示本主题前面部分中演示的查询示例：

```
// Use of var is optional here and in all queries.
// queryCities is an IEnumerable<City> just as
// when it is explicitly typed.
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

有关更多信息，请参见隐式类型的局部变量（C# 编程指南）和 Type Relationships in LINQ Query Operations (C#)。

开始查询表达式

查询表达式必须以 **from** 子句开头。它同时指定了数据源和范围变量。在对源序列进行遍历的过程中，范围变量表示源序列中的每个后续元素。将根据数据源中元素的类型对范围变量进行强类型化。在下面的示例中，因为 countries 是 Country 对象数组，所以范围变量也被类型化为 Country，这样就可以使用点运算符来访问该类型的任何可用成员。

```
IEnumerable<Country> countryAreaQuery =
    from country in countries
    where country.Area > 500000 //sq km
    select country;
```

在使用分号或延续子句退出查询之前，范围变量将一直位于范围中。

查询表达式可以包含多个 **from** 子句。当源序列中的每个元素本身就是集合或包含集合时，可使用附加的 **from** 子句。例如，假定您具有一个 Country 对象集合，而其中每个对象都包含一个名为 Cities 的 City 对象集合。若要查询每个 Country 中的 City 对象，请使用两个**from** 子句，如下所示：

```
IEnumerable<City> cityQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
```

有关更多信息，请参见 from 子句（C# 参考）。

结束查询表达式

查询表达式必须以 **select** 子句或 **group** 子句结尾。

group 子句

使用 **group** 子句可产生按照指定的键组织的组序列。键可以采用任何数据类型。例如，下面的查询创建一个组序列，该序列包含一个或多个 Country 对象，并且它的键是 **char** 值。

```
var queryCountryGroups =
    from country in countries
    group country by country.Name[0];
```

有关分组的更多信息，请参见 group 子句（C# 参考）。

select 子句

使用 **select** 子句可产生所有其他类型的序列。简单的 **select** 子句只是产生与数据源中包含的对象具有相同类型的对象的序列。在此示例中，数据源包含 Country 对象。 **orderby** 子句只是将元素重新排序，而 **select** 子句则产生重新排序的 Country 对象的序列。

```
IEnumerable<Country> sortedQuery =
    from country in countries
    orderby country.Area
    select country;
```

可以使用 **select** 子句将源数据转换为新类型的序列。这一转换也称为“投影”。在下面的示例中，**select** 子句对一个匿名类型序列进行投影，该序列仅包含原始元素中各字段的子集。请注意，新对象是使用对象初始值设定项初始化的。

```
// Here var is required because the query
// produces an anonymous type.
var queryNameAndPop =
    from country in countries
    select new { Name = country.Name, Pop = country.Population };
```

有关使用 **select** 子句转换源数据的所有方式的更多信息，请参见 select 子句（C# 参考）。

使用“into”进行延续

可以在 **select** 或 **group** 子句中使用 **into** 关键字来创建用于存储查询的临时标识符。当您必须在分组或选择操作之后对查询执行附加查询操作时，需要这样做。在下面的示例中，以一千万人口范围为界对 countries 进行分组。在创建这些组之后，使用附加子句筛选掉某些组，然后按升序对剩下的组进行排序。若要执行这些附加操作，需要使用由 countryGroup 表示的延续。

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int) country.Population / 10000000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
        Console.WriteLine(country.Name + ":" + country.Population);
}
```

有关更多信息，请参见 into（C# 参考）。

筛选、排序和联接

在 **from** 开始子句以及 **select** 或 **group** 结束子句之间，所有其他子句（**where**、**join**、**orderby**、**from**、**let**）都是可选的。任何可选子句都可以在查询正文中使用零次或多次。

where 子句

使用 **where** 子句可以根据一个或多个谓词表达式筛选掉源数据中的某些元素。以下示例中的 **where** 子句含有两个谓词。

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population < 200000 && city.Population > 100000
    select city;
```

有关更多信息，请参见 where 子句（C# 参考）。

orderby 子句

使用 **orderby** 子句可以按升序或降序对结果进行排序。您还可以指定次要排序顺序。下面的示例使用 Area 属性对 country 对象执行主要排序，然后使用 Population 属性执行次要排序。

```
IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

ascending 关键字是可选的；如果未指定顺序，则它是默认排序顺序。有关更多信息，请参见 orderby 子句（C# 参考）。

join 子句

使用 **join** 子句可以根据每个元素中指定键之间的相等比较，对一个数据源中的元素与另外一个数据源中的元素进行关联和/或组合。在 LINQ 中，联接操作是针对其元素具有不同类型的对象序列执行的。在联接两个序列之后，必须使用 **select** 或 **group** 语句指定要存储到输出序列中的元素。还可以使用匿名类型将每组关联元素中的属性组合为输出序列的新类型。下面的示例对其 Category 属性与 categories 字符串数组中的某个类别相匹配的 prod 对象进行关联。其 Category 不与 categories 中的任何字符串匹配的产品会被筛选掉。 **select** 语句投影了一个新类型，其属性取自 cat 和 prod。

```
var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new { Category = cat, Name = prod.Name };
```

通过使用 into 关键字将 **join** 操作的结果存储到临时变量中，还可以执行分组联接。有关更多信息，请参见 join 子句（C# 参考）。

let 子句

使用 **let** 子句可以将表达式（如方法调用）的结果存储到新的范围变量中。在下面的示例中，范围变量 firstName 存储了 Split 返回的字符串数组的第一个元素。

```
string[] names = { "Svetlana Omelchenko", "Claire O'Donnell", "Sven
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(new char[] { ' ' })[0]
    select firstName;

foreach (string s in queryFirstNames)
    Console.Write(s + " ");
//Output: Svetlana Claire Sven Cesar
```

有关更多信息，请参见 let 子句（C# 参考）。

查询表达式中的子查询

查询子句本身可能包含一个查询表达式，该查询表达式有时称为“子查询”。每个子查询都以它自己的 **from** 子句开头，该子句不一定指向第一个 **from** 子句中的同一数据源。例如，下面的查询演示了一个在 select 语句中使用的查询表达式，用来检索分组操作的结果。

```
var queryGroupMax =
    from student in students
    group student by student.GradeLevel into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore =
            (from student2 in studentGroup
             select student2.Scores.Average())
             .Max()
    };
```

有关更多信息，请参见 如何：对分组操作执行子查询（C# 编程指南）。

请参阅

C# 编程指南

LINQ 查询表达式（C# 编程指南）

LINQ (Language-Integrated Query)

查询关键字（C# 参考）

Standard Query Operators Overview

如何：在 C# 中编写 LINQ 查询

本主题演示在 C# 中编写 LINQ 查询的三种方式：

1. 使用查询语法。
2. 使用方法语法。
3. 组合使用查询语法和方法语法。

下面的示例使用前面列出的每种方式演示一些简单的 LINQ 查询。一般的规则是尽可能使用 (1)，而在必要时使用 (2) 和 (3)。

注意
这些查询作用于简单的内存中集合；但是，基本语法与 LINQ to SQL 和 LINQ to XML 中使用的语法相同。

查询语法

编写大多数查询的推荐方式是使用查询语法来创建查询表达式。下面的示例演示了三个查询表达式。第一个查询表达式演示如何通过用 **where** 子句应用条件来筛选或限制结果，它返回源序列中值大于 7 或小于 3 的所有元素。第二个表达式演示如何对返回的结果进行排序。第三个表达式演示如何按照键对结果进行分组，此查询可根据单词的第一个字母返回两个组。

```
// Query #1.
List<int> numbers = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0

// The query variable can also be implicitly typed by using var
IEnumerable<int> filteringQuery =
    from num in numbers
    where num < 3 || num > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num < 3 || num > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = { "carrots", "cabbage", "broccoli", "beans
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

请注意，这些查询的类型是 IEnumerable<T>。所有这些查询都可以使用 **var** 编写，如下面的示例所示：

var query = from num in numbers...

在上述每个示例中，直到您在 **foreach** 语句中循环访问查询变量时，查询才会实际执行。有关更多信息，请参见 Introduction to LINQ Queries (C#)。

方法语法

某些查询操作必须表示为方法调用。最常见的此类方法是那些返回单一数值的方法，如 Sum、Max、Min、Average 等。这些方法在任何查询中都必须总是最后调用，因为它们仅表示单个值，不能充当其他查询操作的数据源。下面的示例演示查询表达式中的方法调用：

```
List<int> numbers1 = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, (
List<int> numbers2 = new List<int>() { 15, 14, 11, 13, 19, 18, 16,
// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

如果该方法具有参数，则这些参数以 lambda 表达式的形式提供，如下面的示例所示：

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

在上述查询中，只有查询 4 立即执行。这是因为它返回单个值，而不是一个泛型 IEnumerable<T> 集合。方法本身必须使用 **foreach** 才能计算它的值。

上述每个查询都可以通过结合使用隐式类型化与 var 进行编写，如下面的示例所示：

```
// var is used for convenience in these queries
var average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

混合的查询和方法语法

此示例演示如何对查询子句的结果使用方法语法。只需将查询表达式括在括号内，然后应用点运算符并调用此方法。在下面的示例中，查询 7 返回其值在 3 和 7 之间的数字个数。但是，通常更好的做法是使用另一个变量来存储方法调用的结果。这样就不太容易将查询本身与查询结果相混淆。

```
// Query #7.

// Using a query expression with method syntax
int numCount1 =
    (from num in numbers1
     where num < 3 || num > 7
     select num).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int numCount2 = numbersQuery.Count();
```

由于查询 7 返回单个值而不是一个集合，因此该查询立即执行。

上述查询可以通过结合使用隐式类型化与 **var** 进行编写，如下所示：

```
var numCount = (from num in numbers...
```

它可以按如下方式使用方法语法进行编写：

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

它可以按如下方式使用显式类型化进行编写：

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

请参阅

LINQ (Language-Integrated Query)

Walkthrough: Writing Queries in C# (LINQ)

LINQ 查询表达式（C# 编程指南）

where 子句（C# 参考）

如何：查询对象集合（C# 编程指南）

此示例演示如何对一个 Student 对象列表执行简单查询。每个 Student 对象都包含一些有关该学生的基本信息，以及一个表示该学生的四次考试得分的列表。

此应用程序充当此部分中其他很多示例的框架，这些示例都使用相同的 students 数据源。

下面的查询返回那些在其第一次考试中得分为 90 分或更高的学生。

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYea
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID =
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID =
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID =
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID =
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", ID =
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell",
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 75, 84, 91, 39}},
```

```
        new Student {FirstName = "Svetlana", LastName = "Omelchenko
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", ID =
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", ID
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 94, 92, 91, 91}},
        new Student {FirstName = "Eugene", LastName = "Zabokritski"
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 96, 85, 91, 60}}
    };
    #endregion

    //Helper method, used in GroupByRange.
    protected static int GetPercentile(Student s)
    {
        double avg = s.ExamScores.Average();
        return avg > 0 ? (int)avg / 10 : 0;
    }

    public void QueryHighScores(int exam, int score)
    {
        var highScores = from student in students
                         where student.ExamScores[exam] > score
                         select new {Name = student.FirstName, Scor

        foreach (var item in highScores)
        {
            Console.WriteLine("{0,-15}{1}", item.Name, item.Score);
        }
    }
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

我们特意将此查询编写得非常简单，以使您能够进行试验。例如，您可以在 **where** 子句中尝试使用更多的谓词，也可以使用 **orderby** 子句对结果进行排序。

编译代码

- 创建面向 .NET Framework 3.5 版的 Visual Studio 项目。默认情况下，该项目具有一个对 System.Core.dll 的引用以及一条针对 System.Linq 命名空间的 **using** 指令。
- 将代码复制到项目中。
- 按 F5 编译并运行程序。
- 按任意键退出控制台窗口。

请参阅

LINQ 查询表达式（C# 编程指南）

如何：从方法中返回查询（C# 编程指南）

此示例演示如何从某一方法以返回值和 **out** 参数的形式返回查询。

任何查询的类型都必须为 IEnumerable 或 IEnumerable<T>，或一种派生类型（如 IQueryable<T>）。因此，返回查询的方法的任何返回值或 **out** 参数也必须具有该类型。如果某个方法将查询具体化为具体的 List<T> 或 Array 类型，则认为该方法在返回查询结果（而不是查询本身）。仍然能够编写或修改从方法返回的查询变量。

在下面的示例中，第一个方法以返回值的形式返回查询，第二个方法以 **out** 参数的形式返回查询。请注意，在这两种情况下，返回的都是查询，而不是查询结果。

```
class MQ
{
    // QueryMethhod1 returns a query as its value.
    IEnumerable<string> QueryMethod1(ref int[] ints)
    {
        var intsToStrings = from i in ints
                            where i > 4
                            select i.ToString();
        return intsToStrings;
    }

    // QueryMethod2 returns a query as the value of parameter retur
    void QueryMethod2(ref int[] ints, out IEnumerable<string> retur
    {
        var intsToStrings = from i in ints
                            where i < 4
                            select i.ToString();
        returnQ = intsToStrings;
    }

    static void Main()
    {
        MQ app = new MQ();

        int[] nums = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // QueryMethod1 returns a query as the value of the method
        var myQuery1 = app.QueryMethod1(ref nums);

        // Query myQuery1 is executed in the following foreach loop
        Console.WriteLine("Results of executing myQuery1:");
        // Rest the mouse pointer over myQuery1 to see its type.
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }
```

```
        // You also can execute the query returned from QueryMethoc
        // directly, without using myQuery1.
        Console.WriteLine("\nResults of executing myQuery1 directly
        // Rest the mouse pointer over the call to QueryMethod1 to
        // return type.
        foreach (string s in app.QueryMethod1(ref nums))
        {
            Console.WriteLine(s);
        }

        IEnumerable<string> myQuery2;
        // QueryMethod2 returns a query as the value of its out par
        app.QueryMethod2(ref nums, out myQuery2);

        // Execute the returned query.
        Console.WriteLine("\nResults of executing myQuery2:");
        foreach (string s in myQuery2)
        {
            Console.WriteLine(s);
        }

        // You can modify a query by using query composition. A sav
        // is nested inside a new query definition that revises the
        // of the first query.
        myQuery1 = from item in myQuery1
                   orderby item descending
                   select item;

        // Execute the modified query.
        Console.WriteLine("\nResults of executing modified myQuery1
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

编译代码

- 创建面向 .NET Framework 3.5 或更高版本的 Visual Studio 项目。默认情况下，该项目具有一个对 System.Core.dll 的引用以及一条针对 System.Linq 命名空间的 **using** 指令。
- 将类替换为示例中的代码。

- 按 F5 编译并运行程序。
- 按任意键退出控制台窗口。

请参阅

LINQ 查询表达式（C# 编程指南）

如何：在内存中存储查询结果（C# 编程指南）

查询基本上是一组有关如何检索和组织数据的指令。若要执行查询，需要调用它的 GetEnumerator 方法。当您使用 **foreach** 循环来循环访问元素时，将执行此调用。若要计算查询和存储其结果，而不执行 **foreach** 循环，请对查询变量调用下列方法之一：

- ToList<TSource>
- ToArray<TSource>
- ToDictionary<TSource, TKey, TElement>
- ToLookup<TSource, TKey, TElement>

建议在存储查询结果时，将返回的集合对象分配给一个新变量，如下面的示例所示：

```
class StoreQueryResults
{
    static List<int> numbers = new List<int>() { 1, 2, 4, 6, 8, 10,
    static void Main()
    {

        IEnumerable<int> queryFactorsOfFour =
            from num in numbers
            where num % 4 == 0
            select num;

        // Store the results in a new variable
        // without executing a foreach loop.
        List<int> factorsofFourList = queryFactorsOfFour.ToList();

        // Iterate the list just to prove it holds data.
        foreach (int n in factorsofFourList)
        {
            Console.WriteLine(n);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }
}
```

编译代码

- 创建面向 .NET Framework 3.5 版的 Visual Studio 项目。默认情况下，该项目具有一个对 System.Core.dll 的引用以及一条针对 System.Linq 命名空间的 **using** 指令。
- 将代码复制到项目中。
- 按 F5 编译并运行程序。
- 按任意键退出控制台窗口。

请参阅

LINQ 查询表达式（C# 编程指南）

如何：对查询结果进行分组（C# 编程指南）

分组是 LINQ 最强大的功能之一。下面的示例演示如何以各种方式对数据进行分组：

- 按照单个属性。
- 按照字符串属性的首字母。
- 按照计算出的数值范围。
- 按照布尔谓词或其他表达式。
- 按照复合键。

此外，最后两个查询将它们的结果投影到一个新的匿名类型中，该类型仅包含学生的名字和姓氏。有关更多信息，请参见 group 子句（C# 参考）。

本主题中的所有示例都使用下列帮助器类和数据源。

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYea
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID =
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID =
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID =
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID =
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
```

```
        new Student {FirstName = "Hugo", LastName = "Garcia", ID =
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell",
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", ID =
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", ID
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 94, 92, 91, 91}},
        new Student {FirstName = "Eugene", LastName = "Zabokritski"
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 96, 85, 91, 60}}
    };
    #endregion

    //Helper method, used in GroupByRange.
    protected static int GetPercentile(Student s)
    {
        double avg = s.ExamScores.Average();
        return avg > 0 ? (int)avg / 10 : 0;
    }

    public void QueryHighScores(int exam, int score)
    {
        var highScores = from student in students
                         where student.ExamScores[exam] > score
                         select new {Name = student.FirstName, Scor

        foreach (var item in highScores)
        {
            Console.WriteLine("{0,-15}{1}", item.Name, item.Score);
        }
    }
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
```

```
            Console.ReadKey();
        }
    }
```

下面的示例演示如何通过使用元素的单个属性作为组键对源元素进行分组。在这种情况下，键是 **string**，即学生的姓氏。还可以使用子字符串作为键。分组操作将对该类型使用默认的相等比较器。

将下面的方法粘贴到 StudentClass 类中。将 Main 方法中的调用语句更改为 sc.GroupBySingleProperty()。

```
public void GroupBySingleProperty()
{
    Console.WriteLine("Group by a single property in an object:");

    // Variable queryLastNames is an IEnumerable<IGrouping<string,
    // DataClass.Student>>.
    var queryLastNames =
        from student in students
        group student by student.LastName into newGroup
        orderby newGroup.Key
        select newGroup;

    foreach (var nameGroup in queryLastNames)
    {
        Console.WriteLine("Key: {0}", nameGroup.Key);
        foreach (var student in nameGroup)
        {
            Console.WriteLine("\t{0}, {1}", student.LastName, stude
        }
    }
}
/* Output:
    Group by a single property in an object:
    Key: Adams
            Adams, Terry
    Key: Fakhouri
            Fakhouri, Fadi
    Key: Feng
            Feng, Hanying
    Key: Garcia
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: Mortensen
            Mortensen, Sven
    Key: O'Donnell
            O'Donnell, Claire
    Key: Omelchenko
            Omelchenko, Svetlana
    Key: Tucker
            Tucker, Lance
            Tucker, Michael
    Key: Zabokritski
            Zabokritski, Eugene
*/
```

下面的示例演示如何通过使用除对象属性以外的某个项作为组键对源元素进行分组。在此示例中，键是学生姓氏的第一个字母。

将下面的方法粘贴到 StudentClass 类中。将 Main 方法中的调用语句更改为 sc.GroupBySubstring()。

```
public void GroupBySubstring()
{
    Console.WriteLine("\r\nGroup by something other than a property

    var queryFirstLetters =
        from student in students
        group student by student.LastName[0];

    foreach (var studentGroup in queryFirstLetters)
    {
        Console.WriteLine("Key: {0}", studentGroup.Key);
        // Nested foreach is required to access group items.
        foreach (var student in studentGroup)
        {
            Console.WriteLine("\t{0}, {1}", student.LastName, stude
        }
    }
}
/* Output:
    Group by something other than a property of the object:
    Key: A
            Adams, Terry
    Key: F
            Fakhouri, Fadi
            Feng, Hanying
    Key: G
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: M
            Mortensen, Sven
    Key: O
            O'Donnell, Claire
            Omelchenko, Svetlana
    Key: T
            Tucker, Lance
            Tucker, Michael
    Key: Z
            Zabokritski, Eugene
*/
```

下面的示例演示如何通过使用某个数值范围作为组键对源元素进行分组。然后，查询将结果投影到一个匿名类型中，该类型仅包含学生的名字和姓氏以及该学生所属的百分点范围。使用匿名类型的原因是没有必要使用完整的 Student 对象来显示结果。 GetPercentile 是一个 helper 函数，它根据学生的平均分数计算百分比。该方法返回 0 到 10 之间的整数。

```
//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}
```

将下面的方法粘贴到 StudentClass 类中。将 Main 方法中的调用语句更改为 sc.GroupByRange()。

```
public void GroupByRange()
{
    Console.WriteLine("\r\nGroup by numeric range and project into

    var queryNumericRange =
        from student in students
        let percentile = GetPercentile(student)
        group new { student.FirstName, student.LastName } by percen
        orderby percentGroup.Key
        select percentGroup;

    // Nested foreach required to iterate over groups and group ite
    foreach (var studentGroup in queryNumericRange)
    {
        Console.WriteLine("Key: {0}", (studentGroup.Key * 10));
        foreach (var item in studentGroup)
        {
            Console.WriteLine("\t{0}, {1}", item.LastName, item.Fi
        }
    }
}
/* Output:
    Group by numeric range and project into a new anonymous type:
    Key: 60
            Garcia, Debra
    Key: 70
            O'Donnell, Claire
    Key: 80
            Adams, Terry
            Feng, Hanying
            Garcia, Cesar
            Garcia, Hugo
            Mortensen, Sven
            Omelchenko, Svetlana
            Tucker, Lance
            Zabokritski, Eugene
    Key: 90
            Fakhouri, Fadi
            Tucker, Michael
*/
```

下面的示例演示如何通过使用布尔比较表达式对源元素进行分组。在此示例中，布尔表达式会测试学生的平均考试分数是否超过 75。与上述示例一样，结果被投影到一个匿名类型中，因为不需要完整的源元素。请注意，在执行查询时，该匿名类型中的属性将变成 Key 成员上的属性，并且可以通过名称进行访问。

将下面的方法粘贴到 StudentClass 类中。将 Main 方法中的调用语句更改为 sc.GroupByBoolean()。

```
public void GroupByBoolean()
{
    Console.WriteLine("\r\nGroup by a Boolean into two groups with
    Console.WriteLine("\"True\" and \"False\" and project into a ne
    var queryGroupByAverages = from student in students
                               group new { student.FirstName, stude
                                   by student.ExamScores.Average()
                               select studentGroup;

    foreach (var studentGroup in queryGroupByAverages)
    {
        Console.WriteLine("Key: {0}", studentGroup.Key);
        foreach (var student in studentGroup)
            Console.WriteLine("\t{0} {1}", student.FirstName, stude
    }
}
/* Output:
    Group by a Boolean into two groups with string keys
    "True" and "False" and project into a new anonymous type:
    Key: True
            Terry Adams
            Fadi Fakhouri
            Hanying Feng
            Cesar Garcia
            Hugo Garcia
            Sven Mortensen
            Svetlana Omelchenko
            Lance Tucker
            Michael Tucker
            Eugene Zabokritski
    Key: False
            Debra Garcia
            Claire O'Donnell
*/
```

下面的示例演示如何使用匿名类型来封装包含多个值的键。在此示例中，第一个键值是学生姓氏的第一个字母。第二个键值是一个布尔值，它指定该学生在第一次考试中的得分是否超过了 85。可以按照该键中的任何属性对多组值进行排序。

将下面的方法粘贴到 StudentClass 类中。将 Main 方法中的调用语句更改为 sc.GroupByCompositeKey()。

```
public void GroupByCompositeKey()
{

    var queryHighScoreGroups =
        from student in students
        group student by new { FirstLetter = student.LastName[0],
            Score = student.ExamScores[0] > 85 } into studentGroup
        orderby studentGroup.Key.FirstLetter
        select studentGroup;

    Console.WriteLine("\r\nGroup and order by a compound key:");
    foreach (var scoreGroup in queryHighScoreGroups)
    {
        string s = scoreGroup.Key.Score == true ? "more than" : "le
        Console.WriteLine("Name starts with {0} who scored {1} 85",
        foreach (var item in scoreGroup)
        {
            Console.WriteLine("\t{0} {1}", item.FirstName, item.Las
        }
    }
}
/* Output:
    Group and order by a compound key:
    Name starts with A who scored more than 85
            Terry Adams
    Name starts with F who scored more than 85
            Fadi Fakhouri
            Hanying Feng
    Name starts with G who scored more than 85
            Cesar Garcia
            Hugo Garcia
    Name starts with G who scored less than 85
            Debra Garcia
    Name starts with M who scored more than 85
            Sven Mortensen
    Name starts with O who scored less than 85
            Claire O'Donnell
    Name starts with O who scored more than 85
            Svetlana Omelchenko
    Name starts with T who scored less than 85
            Lance Tucker
    Name starts with T who scored more than 85
            Michael Tucker
    Name starts with Z who scored more than 85
            Eugene Zabokritski
*/
```

编译代码

将您想要测试的每种方法都复制并粘贴到 StudentClass 类中。将该方法的调用语句添加到 Main 方法中并按 F5。

在改写这些方法以适合您自己的应用程序时，请记住 LINQ 需要 .NET Framework 3.5 或 4 版，并且项目必须包含一个对 System.Core.dll 的引用和一条针对 System.Linq 的 using 指令。LINQ to SQL、LINQ to XML 和 LINQ to DataSet 类型需要其他 using 指令和引用。有关更多信息，请参见 How to: Create a LINQ Project。

请参阅

GroupBy<TSource, TKey>

IGrouping<TKey, TElement>

LINQ 查询表达式（C# 编程指南）

group 子句（C# 参考）

匿名类型（C# 编程指南）

如何：对分组操作执行子查询（C# 编程指南）

如何：创建嵌套组（C# 编程指南）

Grouping Data

如何：创建嵌套组（C# 编程指南）

下面的示例演示如何在 LINQ 查询表达式中创建嵌套组。对根据学生年级创建的每个组，将根据每个人的姓名进一步划分为小组。

```
public void QueryNestedGroups()
{
    var queryNestedGroups =
        from student in students
        group student by student.Year into newGroup1
        from newGroup2 in
            (from student in newGroup1
             group student by student.LastName)
        group newGroup2 by newGroup1.Key;

    // Three nested foreach loops are required to iterate
    // over all elements of a grouped group. Hover the mouse
    // cursor over the iteration variables to see their actual type
    foreach (var outerGroup in queryNestedGroups)
    {
        Console.WriteLine("DataClass.Student Level = {0}", outerGro
        foreach (var innerGroup in outerGroup)
        {
            Console.WriteLine("\tNames that begin with: {0}", inner
            foreach (var innerGroupElement in innerGroup)
            {
                Console.WriteLine("\t\t{0} {1}", innerGroupElement.
            }
        }
    }
}
/*
 Output:
DataClass.Student Level = SecondYear
        Names that begin with: Adams
                Adams Terry
        Names that begin with: Garcia
                Garcia Hugo
        Names that begin with: Omelchenko
                Omelchenko Svetlana
DataClass.Student Level = ThirdYear
        Names that begin with: Fakhouri
                Fakhouri Fadi
        Names that begin with: Garcia
                Garcia Debra
        Names that begin with: Tucker
                Tucker Lance
DataClass.Student Level = FirstYear
        Names that begin with: Feng
```

```
                Feng Hanying
        Names that begin with: Mortensen
                Mortensen Sven
        Names that begin with: Tucker
                Tucker Michael
DataClass.Student Level = FourthYear
        Names that begin with: Garcia
                Garcia Cesar
        Names that begin with: O'Donnell
                O'Donnell Claire
        Names that begin with: Zabokritski
                Zabokritski Eugene
 */
```

请注意，需要使用三个嵌套的 **foreach** 循环来循环访问嵌套组的内部元素。

编译代码

此示例包含对如何：查询对象集合（C# 编程指南）内示例应用程序中定义的对象的引用。若要编译和运行此方法，请将它粘贴到该应用程序的 StudentClass 类中，并且在 Main 方法中添加对它的调用。

在改写此方法以适合您自己的应用程序时，请记住 LINQ 需要 .NET Framework 3.5 版，并且项目必须包含一个对 System.Core.dll 的引用和一条针对 System.Linq 的 using 指令。LINQ to SQL、LINQ to XML 和 LINQ to DataSet 类型需要其他 using 指令和引用。有关更多信息，请参见 How to: Create a LINQ Project。

请参阅

LINQ 查询表达式（C# 编程指南）

如何：对分组操作执行子查询（C# 编程指南）

本主题演示执行以下任务的两种不同方式：创建一个查询，以便将源数据排序到不同的组中，然后分别对每个组执行子查询。每个示例中的基本技术都是使用一个名为 newGroup 的“延续”对源元素进行分组，然后生成一个针对 newGroup 的新的子查询。此子查询是针对外部查询所创建的每个新组运行的。请注意，在此特定示例中，最终的输出不是组，而是匿名类型的平面序列。

有关如何分组的更多信息，请参见group 子句（C# 参考）。

有关延续的更多信息，请参见 into（C# 参考）。下面的示例使用内存中数据结构作为数据源，但相同的原理适用于任何种类的 LINQ 数据源。

```
public void QueryMax()
{
    var queryGroupMax =
        from student in students
        group student by student.Year into studentGroup
        select new
        {
            Level = studentGroup.Key,
            HighestScore =
            (from student2 in studentGroup
             select student2.ExamScores.Average()).Max()
        };

    int count = queryGroupMax.Count();
    Console.WriteLine("Number of groups = {0}", count);

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine("  {0} Highest Score={1}", item.Level, it
    }
}
```

编译代码

此示例包含对如何：查询对象集合（C# 编程指南）内示例应用程序中定义的对象的引用。若要编译和运行此方法，请将它粘贴到该应用程序的 StudentClass 类中，并且在 Main 方法中添加对它的调用。

在改写此方法以适合您自己的应用程序时，请记住 LINQ 需要 .NET Framework 3.5 版，并且项目必须包含一个对 System.Core.dll 的引用和一条针对 System.Linq 的 using 指令。LINQ to SQL、LINQ to XML 和 LINQ to DataSet 类型需要其他 using 指令和引用。有关更多信息，请参见 How to: Create a LINQ Project。

请参阅

LINQ 查询表达式（C# 编程指南）

如何：按连续键对结果进行分组（C# 编程指南）

下面的示例演示如何将元素归到不同的块区中，块区表示连续键的子序列。例如，假设您有下面的键值对序列：

键	值
A	We
A	think
A	that
B	Linq
C	is
A	really
B	cool
B	!

将按照下面的顺序创建以下各组：

1. We, think, that
2. Linq
3. is
4. really
5. cool, !

此解决方案作为扩展方法实现，扩展方法是线程安全的且以流的方式返回其结果。也就是说，它在移动通过源序列的过程中生成自己的各个组。与 **group** 或 **orderby** 运算符不同，它可以在整个序列读取完毕之前就开始将组返回给调用方。

线程安全性是通过在迭代源序列的过程中创建每个组或块区的副本来实现的，如源代码注释中所述。如果源序列包含一个由连续项组成的很大的序列，则公共语言运行时可能会引发 OutOfMemoryException。

下面的示例显示了此扩展方法及使用它的客户端代码。

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ChunkIt
{
```

```
    // Static class to contain the extension methods.
    public static class MyExtensions
    {
        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy
        {
            return source.ChunkBy(keySelector, EqualityComparer<TKe
        }

        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy
        {
            // Flag to signal end of source sequence.
            const bool noMoreSourceElements = true;

            // Auto-generated iterator for the source array.
            var enumerator = source.GetEnumerator();

            // Move to the first element in the source sequence.
            if (!enumerator.MoveNext()) yield break;

            // Iterate through source sequence and create a copy of
            // On each pass, the iterator advances to the first ele
            // in the source sequence. This loop corresponds to the
            // executes the query.
            Chunk<TKey, TSource> current = null;
            while (true)
            {
                // Get the key for the current Chunk. The source it
                // the source sequence until it finds an element wi
                var key = keySelector(enumerator.Current);

                // Make a new Chunk (group) object that initially h
                current = new Chunk<TKey, TSource>(key, enumerator,

                // Return the Chunk. A Chunk is an IGrouping<TKey,T
                // At this point the Chunk only has the first eleme
                // returned only when the client code foreach's ove
                yield return current;

                // Check to see whether (a) the chunk has made a co
                // (b) the iterator has reached the end of the sour
                // foreach loop to iterate the chunk items, and tha
                // then the Chunk.GetEnumerator method will already
                // copies of all chunk items before we get here. If
                // enumerate all elements in the chunk, we need to
                // for clients that may be calling us on a separate
                if (current.CopyAllChunkElements() == noMoreSourceE
                {
                    yield break;
                }
            }
        }

        // A Chunk is a contiguous group of one or more source elem
```

```
        // has a key and a list of ChunkItem objects, which are cop
        class Chunk<TKey, TSource> : IGrouping<TKey, TSource>
        {
            // INVARIANT: DoneCopyingChunk == true ||
            //   (predicate != null && predicate(enumerator.Current

            // A Chunk has a linked list of ChunkItems, which repre
            // has a reference to the next ChunkItem in the list.
            class ChunkItem
            {
                public ChunkItem(TSource value)
                {
                    Value = value;
                }
                public readonly TSource Value;
                public ChunkItem Next = null;
            }
            // The value that is used to determine matching element
            private readonly TKey key;

            // Stores a reference to the enumerator for the source
            private IEnumerator<TSource> enumerator;

            // A reference to the predicate that is used to compare
            private Func<TSource, bool> predicate;

            // Stores the contents of the first source element that
            // belongs with this chunk.
            private readonly ChunkItem head;

            // End of the list. It is repositioned each time a new
            // ChunkItem is added.
            private ChunkItem tail;

            // Flag to indicate the source iterator has reached the
            internal bool isLastSourceElement = false;

            // Private object for thread syncronization
            private object m_Lock;

            // REQUIRES: enumerator != null && predicate != null
            public Chunk(TKey key, IEnumerator<TSource> enumerator,
            {
                this.key = key;
                this.enumerator = enumerator;
                this.predicate = predicate;

                // A Chunk always contains at least one element.
                head = new ChunkItem(enumerator.Current);

                // The end and beginning are the same until the lis
                tail = head;
```

```
                m_Lock = new object();
            }

            // Indicates that all chunk elements have been copied t
            // and the source enumerator is either at the end, or e
            // the tail of the linked list is set to null in the Co
            // key of the next element does not match the current c
            private bool DoneCopyingChunk { get { return tail == nu

            // Adds one ChunkItem to the current group
            // REQUIRES: !DoneCopyingChunk && lock(this)
            private void CopyNextChunkElement()
            {
                // Try to advance the iterator on the source sequer
                // If MoveNext returns false we are at the end, anc
                isLastSourceElement = !enumerator.MoveNext();

                // If we are (a) at the end of the source, or (b) a
                // then null out the enumerator and predicate for r
                if (isLastSourceElement || !predicate(enumerator.Cu
                {
                    enumerator = null;
                    predicate = null;
                }
                else
                {
                    tail.Next = new ChunkItem(enumerator.Current);
                }

                // tail will be null if we are at the end of the ch
                // This check is made in DoneCopyingChunk.
                tail = tail.Next;
            }

            // Called after the end of the last chunk was reached.
            // there are more elements in the source sequence. If t
            // Returns true if enumerator for this chunk was exhaus
            internal bool CopyAllChunkElements()
            {
                while (true)
                {
                    lock (m_Lock)
                    {
                        if (DoneCopyingChunk)
                        {
                            // If isLastSourceElement is false,
                            // it signals to the outer iterator
                            // to continue iterating.
                            return isLastSourceElement;
                        }
                        else
                        {
                            CopyNextChunkElement();
```

```
                    }
                }
            }
        }

        public TKey Key { get { return key; } }

        // Invoked by the inner foreach loop. This method stays
        // of the client requests. It adds the next element of
        // the clients requests the last element in the list so
        public IEnumerator<TSource> GetEnumerator()
        {
            //Specify the initial element to enumerate.
            ChunkItem current = head;

            // There should always be at least one ChunkItem in
            while (current != null)
            {
                // Yield the current item in the list.
                yield return current.Value;

                // Copy the next item from the source sequence,
                // if we are at the end of our local list.
                lock (m_Lock)
                {
                    if (current == tail)
                    {
                        CopyNextChunkElement();
                    }
                }

                // Move to the next ChunkItem in the list.
                current = current.Next;
            }
        }

        System.Collections.IEnumerator System.Collections.IEnum
        {
            return GetEnumerator();
        }
    }
}

// A simple named type is used for easier viewing in the debug
// work just as well with the ChunkBy operator.
public class KeyValPair
{
    public string Key { get; set; }
    public string Value { get; set; }
}

class Program
{
```

```
        // The source sequence.
        public static IEnumerable<KeyValPair> list;

        // Query variable declared as class member to be available
        // on different threads.
        static IEnumerable<IGrouping<string, KeyValPair>> query;

        static void Main(string[] args)
        {
            // Initialize the source sequence with an array initial
            list = new[]
        {
            new KeyValPair{ Key = "A", Value = "We" },
            new KeyValPair{ Key = "A", Value = "Think" },
            new KeyValPair{ Key = "A", Value = "That" },
            new KeyValPair{ Key = "B", Value = "Linq" },
            new KeyValPair{ Key = "C", Value = "Is" },
            new KeyValPair{ Key = "A", Value = "Really" },
            new KeyValPair{ Key = "B", Value = "Cool" },
            new KeyValPair{ Key = "B", Value = "!" }
        };

            // Create the query by using our user-defined query ope
            query = list.ChunkBy(p => p.Key);

            // ChunkBy returns IGrouping objects, therefore a neste
            // foreach loop is required to access the elements in e
            foreach (var item in query)
            {
                Console.WriteLine("Group key = {0}", item.Key);
                foreach (var inner in item)
                {
                    Console.WriteLine("\t{0}", inner.Value);
                }
            }

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }

    }
}
```

若要在您的项目中使用此扩展方法，请将 MyExtensions 静态类复制到一个新的或现有源代码文件中，并在必要时，为该类所在的命名空间添加一条 **using** 指令。

请参阅

LINQ 查询表达式（C# 编程指南）

Classification of Standard Query Operators by Manner of Execution

如何：在运行时动态指定谓词筛选器（C# 编程指南）

在某些情况下，您直到运行时才能知道必须将多少个谓词应用于 **where** 子句中的源元素。动态指定多个谓词筛选器的一种方式是使用 Contains<TSource> 方法，如下面的示例中所示。此示例按两种方式构造。首先，通过筛选程序中提供的值来运行项目。然后，使用运行时提供的输入再次运行项目。

使用 Contains 方法进行筛选

1. 在 Visual Studio 中，打开一个新的控制台应用程序。将其命名为 PredicateFilters。

2. 从如何：查询对象集合（C# 编程指南）中复制 StudentClass 类，并将此类粘贴到 Program 类下方的命名空间 PredicateFilters 中。 StudentClass 提供了 Student 对象的列表。

3. 将 StudentClass 中的 Main 方法注释掉。

4. 将 Program 类替换为以下代码。

```
class DynamicPredicates : StudentClass
{
    static void Main(string[] args)
    {
        string[] ids = { "111", "114", "112" };

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryByID(string[] ids)
    {
        var queryNames =
            from student in students
            let i = student.ID.ToString()
            where ids.Contains(i)
            select new { student.LastName, student.ID };

        foreach (var name in queryNames)
        {
            Console.WriteLine("{0}: {1}", name.LastName, name.I
        }
    }
}
```

5. 在 DynamicPredicates 类的 Main 方法中的 ids 声明下添加以下行。

```
QueryById(ids);
```

6. 按 F5 运行项目。
7. 命令提示符窗口中将显示以下输出：

   Garcia: 114

   O'Donnell: 112

   Omelchenko: 111

8. 下一步是重新运行项目，这次使用在运行时输入的输入内容而不是数组 ids 来运行项目。在“解决方案资源管理器”中右击“**PredicateFilters**”，然后单击“属性”。
9. 单击“调试”选项卡。
10. 在“命令行参数”窗口中，按如下方式键入 122、117、120 和 115（用空格隔开）：**122 117 120 115**。在运行项目时，这些值将成为 Main 方法的参数 args 的元素。
11. 在 Main 方法中，将 QueryByID(ids) 更改为 QueryByID(args)。
12. 按 F5 运行项目。
13. 命令提示符窗口中将显示以下输出：

    Adams: 120

    Feng: 117

    Garcia: 115

    Tucker: 122

使用 switch 语句进行筛选

1. 可以使用 **switch** 语句在预先确定的备选查询中进行选择。在下面的示例中，studentQuery 使用的 **where** 子句随着在运行时指定的年级或年份的不同而不同。
2. 复制以下方法并将其粘贴到类 DynamicPredicates 中。

```
// To run this sample, first specify an integer value of 1 to 4
// line. This number will be converted to a GradeLevel value th
// set of students to query.
// Call the method: QueryByYear(args[0]);

static void QueryByYear(string level)
{
    GradeLevel year = (GradeLevel)Convert.ToInt32(level);
    IEnumerable&lt;Student&gt; studentQuery = null;
    switch (year)
    {
        case GradeLevel.FirstYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.Fir
                           select student;
            break;
        case GradeLevel.SecondYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.Sec
                           select student;
            break;
        case GradeLevel.ThirdYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.Thi
                           select student;
            break;
        case GradeLevel.FourthYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.Fou
                           select student;
            break;

        default:
            break;
    }
    Console.WriteLine("The following students are at level {0}"
    foreach (Student name in studentQuery)
    {
        Console.WriteLine("{0}: {1}", name.LastName, name.ID);
    }
}
```

3. 在“命令行参数”窗口中，将上一过程中的 ID 号替换为一个介于 1 到 4 之间的整数值。

4. 在 Main 方法中，将对 QueryByID 的调用替换为以下调用，此调用可将 args 数组中的第一个元素作为其参数发送：QueryByYear(args[0])。

5. 按 F5 运行项目。

在您自己的应用程序中使用此方法

- 在改编此方法以适合您自己的应用程序时，请记住，LINQ 需要版本 3.5 或版本 4 的 .NET Framework，并且项目必须包含一个对 System.Core.dll 的引用和一条针对 **System.Linq** 的 **using** 指令。LINQ to SQL、LINQ to XML 和 LINQ to DataSet 类型需要相应的其他 **using** 指令和引用。有关更多信息，请参见 How to: Create a LINQ Project。

请参阅

LINQ 查询表达式（C# 编程指南）

where 子句（C# 参考）

如何：执行内部联接（C# 编程指南）

按照关系数据库的说法，“内部联接”产生一个结果集，对于该结果集内第一个集合中的每个元素，只要在第二个集合中存在一个匹配元素，该元素就会出现一次。如果第一个集合中的某个元素没有匹配元素，则它不会出现在结果集内。 Join 方法（通过 C# 中的 **join** 子句调用）可实现内联。

本主题演示如何执行内部联接的四种变体：

- 简单的内部联接，它基于一个简单的键将来自两个数据源的元素相互关联。
- 内部联接，它基于一个复合键将来自两个数据源的元素相互关联。使用复合键（即由多个值组成的键）可以基于多个属性将元素相互关联。
- 多联接，在其中连续的联接操作被相互拼接在一起。
- 通过使用分组联接实现的内部联接。

简单键联接示例

下面的示例创建了两个集合，其中分别包含以下两个用户定义类型的对象：Person 和 Pet。查询使用 C# 中的 **join** 子句来将 Person 对象与 Pet 对象（其 Owner 为该 Person）进行匹配。C# 中的 **select** 子句可定义生成的对象的外观。在此示例中，生成的对象是由主人的名字和宠物的名字组成的匿名类型。

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Simple inner join.
/// </summary>
public static void InnerJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "
    Person terry = new Person { FirstName = "Terry", LastName = "Ad
    Person charlotte = new Person { FirstName = "Charlotte", LastNa
    Person arlene = new Person { FirstName = "Arlene", LastName = "
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo
```

```
        Pet barley = new Pet { Name = "Barley", Owner = terry };
        Pet boots = new Pet { Name = "Boots", Owner = terry };
        Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte ]
        Pet bluemoon = new Pet { Name = "Blue Moon", Owner = rui };
        Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

        // Create two lists.
        List<Person> people = new List<Person> { magnus, terry, charlot
        List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluer

        // Create a collection of person-pet pairs. Each element in the
        // is an anonymous type containing both the person's name and t
        var query = from person in people
                    join pet in pets on person equals pet.Owner
                    select new { OwnerName = person.FirstName, PetName

        foreach (var ownerAndPet in query)
        {
            Console.WriteLine("\"{0}\" is owned by {1}", ownerAndPet.Pe
        }
    }

    // This code produces the following output:
    //
    // "Daisy" is owned by Magnus
    // "Barley" is owned by Terry
    // "Boots" is owned by Terry
    // "Whiskers" is owned by Charlotte
    // "Blue Moon" is owned by Rui
```

请注意，其 LastName 为“Huff”的 Person 对象未出现在结果集内，因为不存在 Pet.Owner 等于该 Person 的 Pet 对象。

复合键联接示例

与仅仅基于一个属性将元素相互关联不同，使用复合键可基于多个属性来比较元素。为此，需要为每个集合指定键选择器函数，以便返回一个由要比较的属性组成的匿名类型。如果给属性加上了标签，则这些属性必须在每个键的匿名类型中都有相同的标签，而且还必须以相同顺序出现。

下面的示例使用一个 Employee 对象列表和一个 Student 对象列表来确定哪些雇员同时还是学生。这两个类型都具有 String 类型的 FirstName 和 LastName 属性。能够从每个列表的元素创建联接键的函数可返回一个由每个元素的 FirstName 和 LastName 属性组成的匿名类型。联接操作比较这些复合键是否相等，并且从每个列表中返回名字和姓氏都匹配的对象对。

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int EmployeeID { get; set; }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int StudentID { get; set; }
}

/// <summary>
/// Performs a join operation using a composite key.
/// </summary>
public static void CompositeKeyJoinExample()
{
    // Create a list of employees.
    List<Employee> employees = new List<Employee> {
        new Employee { FirstName = "Terry", LastName = "Adams", Emp
         new Employee { FirstName = "Charlotte", LastName = "Weiss'
         new Employee { FirstName = "Magnus", LastName = "Hedland",
         new Employee { FirstName = "Vernette", LastName = "Price",

    // Create a list of students.
    List<Student> students = new List<Student> {
        new Student { FirstName = "Vernette", LastName = "Price", S
        new Student { FirstName = "Terry", LastName = "Earls", Stud
        new Student { FirstName = "Terry", LastName = "Adams", Stud

    // Join the two data sources based on a composite key consisti
    // to determine which employees are also students.
    IEnumerable<string> query = from employee in employees
                                join student in students
                                on new { employee.FirstName, employ
                                equals new { student.FirstName, stu
                                select employee.FirstName + " " + e

    Console.WriteLine("The following people are both employees and
    foreach (string name in query)
        Console.WriteLine(name);
}

// This code produces the following output:
//
// The following people are both employees and students:
// Terry Adams
// Vernette Price
```

多联接示例

可以将任意数量的联接操作拼接在一起以执行多联接。C# 中的每一个 **join** 子句都可将指定的数据源与前一个联接的结果相互关联。

下面的示例创建了三个集合：一个 Person 对象列表、一个 Cat 对象列表以及一个 Dog 对象列表。

C# 中的第一个 **join** 子句将基于匹配 Cat.Owner 的 Person 对象对主人和猫进行匹配。并返回包含 Person 对象和 Cat.Name 的匿名类型的序列。

C# 中的第二个 **join** 子句基于一个组合键将第一个联接返回的匿名类型与所提供的犬列表中的 Dog 对象相互关联，该组合键由类型为 Person 的 Owner 属性和动物名字的首字母组成。该子句返回一个匿名类型序列，这些类型包含每个匹配对中的 Cat.Name 和 Dog.Name 属性。由于这是一个内部联接，因此仅返回第一个数据源中那些在第二个数据源中具有匹配对象的对象。

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

class Cat : Pet
{ }

class Dog : Pet
{ }

public static void MultipleJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = '
    Person terry = new Person { FirstName = "Terry", LastName = "Ad
    Person charlotte = new Person { FirstName = "Charlotte", LastNa
    Person arlene = new Person { FirstName = "Arlene", LastName = '
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo
    Person phyllis = new Person { FirstName = "Phyllis", LastName =

    Cat barley = new Cat { Name = "Barley", Owner = terry };
    Cat boots = new Cat { Name = "Boots", Owner = terry };
    Cat whiskers = new Cat { Name = "Whiskers", Owner = charlotte }
    Cat bluemoon = new Cat { Name = "Blue Moon", Owner = rui };
    Cat daisy = new Cat { Name = "Daisy", Owner = magnus };

    Dog fourwheeldrive = new Dog { Name = "Four Wheel Drive", Owner
    Dog duke = new Dog { Name = "Duke", Owner = magnus };
```

```
    Dog denim = new Dog { Name = "Denim", Owner = terry };
    Dog wiley = new Dog { Name = "Wiley", Owner = charlotte };
    Dog snoopy = new Dog { Name = "Snoopy", Owner = rui };
    Dog snickers = new Dog { Name = "Snickers", Owner = arlene };

    // Create three lists.
    List<Person> people =
        new List<Person> { magnus, terry, charlotte, arlene, rui, p
    List<Cat> cats =
        new List<Cat> { barley, boots, whiskers, bluemoon, daisy };
    List<Dog> dogs =
        new List<Dog> { fourwheeldrive, duke, denim, wiley, snoopy,

    // The first join matches Person and Cat.Owner from the list of
    // cats, based on a common Person. The second join matches dogs
    // with the same letter as the cats that have the same owner.
    var query = from person in people
                join cat in cats on person equals cat.Owner
                join dog in dogs on
                new { Owner = person, Letter = cat.Name.Substring(0
                equals new { dog.Owner, Letter = dog.Name.Substring
                select new { CatName = cat.Name, DogName = dog.Name

    foreach (var obj in query)
    {
        Console.WriteLine(
            "The cat \"{0}\" shares a house, and the first letter o
            obj.CatName, obj.DogName);
    }
}

// This code produces the following output:
//
// The cat "Daisy" shares a house, and the first letter of their na
// The cat "Whiskers" shares a house, and the first letter of their
```

使用分组联接实现内部联接的示例

下面的示例演示如何使用分组联接来实现内部联接。

在 query1 中，Person 对象列表基于与 Pet.Owner 属性匹配的 Person 分组联接到 Pet 对象列表。分组联接创建了一个中间组集合，该集合中的每个组都由一个 Person 对象和匹配的 Pet 对象序列组成。

通过向查询中添加另一个 **from** 子句，此序列的序列被组合（或展平）为一个较长的序列。最终序列的元素类型由 **select** 子句指定。在此示例中，该类型是由每个匹配对的 Person.FirstName 和 Pet.Name 属性组成的匿名类型。

query1 的结果等效于使用 **join** 子句所获得的结果集，而没有 **into** 子句执行内联。query2 变量演示了这一等效查询。

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Performs an inner join by using GroupJoin().
/// </summary>
public static void InnerGroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = '
    Person terry = new Person { FirstName = "Terry", LastName = "Ad
    Person charlotte = new Person { FirstName = "Charlotte", LastNa
    Person arlene = new Person { FirstName = "Arlene", LastName = '

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte }
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlot
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluem

    var query1 = from person in people
                 join pet in pets on person equals pet.Owner into g
                 from subpet in gj
                 select new { OwnerName = person.FirstName, PetName

    Console.WriteLine("Inner join using GroupJoin():");
    foreach (var v in query1)
    {
        Console.WriteLine("{0} - {1}", v.OwnerName, v.PetName);
    }

    var query2 = from person in people
                 join pet in pets on person equals pet.Owner
                 select new { OwnerName = person.FirstName, PetName

    Console.WriteLine("\nThe equivalent operation using Join():");
    foreach (var v in query2)
        Console.WriteLine("{0} - {1}", v.OwnerName, v.PetName);
}
```

```
// This code produces the following output:
//
// Inner join using GroupJoin():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
//
// The equivalent operation using Join():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
```

编译代码

- 在 Visual Studio 中创建一个新的控制台应用程序项目。
- 添加对 System.Core.dll 的引用（如果尚未引用它的话）。
- 包含 System.Linq 命名空间。
- 从示例中复制代码，并将其粘贴到 program.cs 文件中的 Main 方法之下。向 Main 方法添加一行代码，以调用粘入的方法。
- 运行该程序。

请参阅

Join

GroupJoin

Join Operations

如何：执行分组联接（C# 编程指南）

如何：执行左外部联接（C# 编程指南）

如何：联接两个集合 (C#) (LINQ to XML)

匿名类型（C# 编程指南）

匿名类型 (Visual Basic)

如何：执行分组联接（C# 编程指南）

分组联接可用于产生分层数据结构。它将第一个集合中的每个元素与第二个集合中的一组相关元素进行配对。

例如，一个名为 Student 的类或关系数据库表可能包含两个字段：Id 和 Name。另一个名为 Course 的类或关系数据库表可能包含两个字段：StudentId 和 CourseTitle。这两个数据源的分组联接（基于匹配的 Student.Id 和 Course.StudentId）会将每个 Student 与一个 Course 对象集合（可能为空）组合在一起。

> 注意
>
> 第一个集合中的每个元素都会出现在分组联接的结果集内，而无论是否在第二个集合中找到相关元素。如果未找到相关元素，则该元素的相关元素序列为空。因此，结果选择器可以访问第一个集合的每个元素。非分组联接中的结果选择器与此不同，它无法访问第一个集合中的那些在第二个集合中没有匹配元素的元素。

本主题中的第一个示例演示如何执行分组联接；第二个示例演示如何使用分组联接创建 XML 元素。

分组联接示例

下面的示例根据与 Pet.Owner 属性匹配的 Person 来对 Person 和 Pet 类型的对象执行分组联接。与为每个匹配产生一对元素的非分组联接不同，分组联接只为第一个集合的每个元素产生一个结果对象（在此示例中为一个 Person 对象）。第二个集合中的相应元素（在此示例中为 Pet 对象）被分组到一个集合中。最后，结果选择器函数为每个包含 Person.FirstName 和一个 Pet 对象集合的匹配创建一个匿名类型。

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example performs a grouped join.
/// </summary>
public static void GroupJoinExample()
```

```
{
    Person magnus = new Person { FirstName = "Magnus", LastName = '
    Person terry = new Person { FirstName = "Terry", LastName = "Ad
    Person charlotte = new Person { FirstName = "Charlotte", LastNa
    Person arlene = new Person { FirstName = "Arlene", LastName = '

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte ]
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlot
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluer

    // Create a list where each element is an anonymous type
    // that contains the person's first name and a collection of
    // pets that are owned by them.
    var query = from person in people
                join pet in pets on person equals pet.Owner into g
                select new { OwnerName = person.FirstName, Pets = g

    foreach (var v in query)
    {
        // Output the owner's name.
        Console.WriteLine("{0}:", v.OwnerName);
        // Output each of the owner's pet's names.
        foreach (Pet pet in v.Pets)
            Console.WriteLine("  {0}", pet.Name);
    }
}

// This code produces the following output:
//
// Magnus:
//   Daisy
// Terry:
//   Barley
//   Boots
//   Blue Moon
// Charlotte:
//   Whiskers
// Arlene:
```

执行分组联接以创建 XML 的示例

分组联接非常适合于使用 LINQ to XML 来创建 XML。下面的示例与上一个示例类似，不同之处在于：结果选择器函数创建表示已联接对象的 XML 元素，而不是创建匿名类型。有关 LINQ to XML的更多信息，请参见LINQ to XML。

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example creates XML output from a grouped join.
/// </summary>
public static void GroupJoinXMLExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = '
    Person terry = new Person { FirstName = "Terry", LastName = "Ad
    Person charlotte = new Person { FirstName = "Charlotte", LastNa
    Person arlene = new Person { FirstName = "Arlene", LastName = '

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte }
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlot
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluer

    // Create XML to display the hierarchical organization of peopl
    XElement ownersAndPets = new XElement("PetOwners",
        from person in people
        join pet in pets on person equals pet.Owner into gj
        select new XElement("Person",
            new XAttribute("FirstName", person.FirstName),
            new XAttribute("LastName", person.LastName),
            from subpet in gj
            select new XElement("Pet", subpet.Name)));

    Console.WriteLine(ownersAndPets);
}

// This code produces the following output:
//
// <PetOwners>
//   <Person FirstName="Magnus" LastName="Hedlund">
//     <Pet>Daisy</Pet>
//   </Person>
```

```
//    <Person FirstName="Terry" LastName="Adams">
//      <Pet>Barley</Pet>
//      <Pet>Boots</Pet>
//      <Pet>Blue Moon</Pet>
//    </Person>
//    <Person FirstName="Charlotte" LastName="Weiss">
//      <Pet>Whiskers</Pet>
//    </Person>
//    <Person FirstName="Arlene" LastName="Huff" />
//  </PetOwners>
```

编译代码

- 在 Visual Studio 中创建一个新的“控制台应用程序”项目。
- 添加一个对 System.Core.dll 的引用和一个对 System.Xml.Linq.dll 的引用（如果它们尚未被引用的话）。
- 包括 System.Linq 和 System.Xml.Linq 命名空间。
- 从示例中复制代码，并将其粘贴到 program.cs 文件中的 Main 方法之下。向 Main 方法添加一行代码，以调用粘入的方法。
- 运行该程序。

请参阅

Join

GroupJoin

Join Operations

如何：执行内部联接（C# 编程指南）

如何：执行左外部联接（C# 编程指南）

LINQ to XML

匿名类型（C# 编程指南）

匿名类型 (Visual Basic)

如何：执行左外部联接（C# 编程指南）

左外部联接是这样一个联接：在其中返回第一个集合的每个元素，而无论该元素在第二个集合中是否具有相关元素。可以使用 LINQ 执行左通过对分组联接的结果调用方法 DefaultIfEmpty<TSource> 外部联接连接。

下面的示例演示如何对分组联接的结果调用 DefaultIfEmpty<TSource> 方法来执行左外部联接。

若要生成两个集合的左外部联接，第一步是使用分组联接执行内部联接。（有关此过程的说明，请参见如何：执行内部联接（C# 编程指南）。）在此示例中，Person 对象的列表内部联接到 Pet 对象列表。Person 对象的匹配 Pet.Owner。

第二步是在结果集内包含第一个（左）集合的每个元素，即使该元素在右集合中没有匹配的元素也是如此。这是通过对分组联接中的每个匹配元素序列调用 DefaultIfEmpty<TSource> 来实现的。在此示例中，DefaultIfEmpty<TSource> 对匹配 Pet 对象每个序列。方法返回一个包含单个集合，因此，如果匹配 Pet 对象序列为任何 Person 对象为空，从而确保的默认值每 Person 对象在结果集合表示。

注意
引用类型的默认值为 **null**;因此，该示例检查空在访问每个 Pet 集合的每个元素之前引用。

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void LeftOuterJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = '
    Person terry = new Person { FirstName = "Terry", LastName = "Ac
    Person charlotte = new Person { FirstName = "Charlotte", LastNa
    Person arlene = new Person { FirstName = "Arlene", LastName = '

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte ]
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlot
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluer

    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                from subpet in gj.DefaultIfEmpty()
                select new { person.FirstName, PetName = (subpet ==

    foreach (var v in query)
    {
        Console.WriteLine("{0,-15}{1}", v.FirstName + ":", v.PetNar
    }
}

// This code produces the following output:
//
// Magnus:         Daisy
// Terry:          Barley
// Terry:          Boots
// Terry:          Blue Moon
// Charlotte:      Whiskers
// Arlene:
```

编译代码

- 在 Visual Studio 中创建一个新的控制台应用程序项目。
- 添加对 System.Core.dll 的引用（如果尚未引用它的话）。
- 包含 System.Linq 命名空间。
- 复制和粘贴该示例的代码添加到 program.cs 文件中，在 Program 选件类的 Main 方法。添加代码行。Main 方法调用 LeftOuterJoinExample 方法。
- 运行该程序。

请参阅

Join

GroupJoin

Join Operations

如何：执行内部联接（C# 编程指南）

如何：执行分组联接（C# 编程指南）

匿名类型（C# 编程指南）

匿名类型 (Visual Basic)

如何：对 Join 子句的结果进行排序（C# 编程指南）

此示例演示如何对联接运算的结果进行排序。请注意，排序是在联接之后执行的。尽管可以在联接之前将 **orderby** 子句用于一个或多个源序列，但通常我们不建议这样做。某些 LINQ 提供程序可能不会在联接之后保留排序。

此查询将创建一个分组联接，然后基于类别元素（仍然在范围中）对组进行排序。在匿名类型初始值设定项中，子查询将对产品序列中的所有匹配元素进行排序。

```
    class HowToOrderJoins
    {
        #region Data
        class Product
        {
            public string Name { get; set; }
            public int CategoryID { get; set; }
        }

        class Category
        {
            public string Name { get; set; }
            public int ID { get; set; }
        }

        // Specify the first data source.
        List<Category> categories = new List<Category>()
{
    new Category(){Name="Beverages", ID=001},
    new Category(){ Name="Condiments", ID=002},
    new Category(){ Name="Vegetables", ID=003},
    new Category() {  Name="Grains", ID=004},
    new Category() {  Name="Fruit", ID=005}
};

        // Specify the second data source.
        List<Product> products = new List<Product>()
{
   new Product{Name="Cola",  CategoryID=001},
   new Product{Name="Tea",  CategoryID=001},
   new Product{Name="Mustard", CategoryID=002},
   new Product{Name="Pickles", CategoryID=002},
   new Product{Name="Carrots", CategoryID=003},
   new Product{Name="Bok Choy", CategoryID=003},
   new Product{Name="Peaches", CategoryID=005},
   new Product{Name="Melons", CategoryID=005},
 };
        #endregion
```

```
        static void Main()
        {
            HowToOrderJoins app = new HowToOrderJoins();
            app.OrderJoin1();

            // Keep console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();

        }

        void OrderJoin1()
        {
            var groupJoinQuery2 =
                from category in categories
                join prod in products on category.ID equals prod.C
                orderby category.Name
                select new
                {
                    Category = category.Name,
                    Products = from prod2 in prodGroup
                               orderby prod2.Name
                               select prod2
                };

            foreach (var productGroup in groupJoinQuery2)
            {
                Console.WriteLine(productGroup.Category);
                foreach (var prodItem in productGroup.Products)
                {
                    Console.WriteLine("  {0,-10} {1}", prodItem.Na
                }
            }
        }
        /* Output:
            Beverages
              Cola       1
              Tea        1
            Condiments
              Mustard    2
              Pickles    2
            Fruit
              Melons     5
              Peaches    5
            Grains
            Vegetables
              Bok Choy   3
              Carrots    3
         */
    }
```

编译代码

- 创建面向 .NET Framework 3.5 版的 Visual Studio 项目。默认情况下，该项目具有一个对 System.Core.dll 的引用以及一条针对 System.Linq 命名空间的 **using** 指令。
- 将代码复制到项目中。
- 按 F5 编译并运行程序。
- 按任意键退出控制台窗口。

请参阅

LINQ 查询表达式（C# 编程指南）

orderby 子句（C# 参考）

join 子句（C# 参考）

Join Operations

如何：使用复合键进行联接（C# 编程指南）

此示例演示如何执行想要使用多个键来定义匹配的联接操作。使用组合键来完成此操作。以匿名类型或包含要比较的值的命名类型的形式来创建组合键。如果将跨方法边界传递查询变量，请使用为该键重写 Equals 和 GetHashCode 的命名类型。属性的名称以及属性出现的顺序在每个键中必须相同。

下面的示例演示如何使用组合键来联接来自三个表中的数据：

```
var query = from o in db.Orders
    from p in db.Products
    join d in db.OrderDetails
        on new {o.OrderID, p.ProductID} equals new {d.OrderID,
        d.ProductID} into details
        from d in details
        select new {o.OrderID, p.ProductID, d.UnitPrice};
```

组合键上的类型推理取决于这些键中属性的名称，以及属性的出现顺序。如果源序列中属性的名称不同，您必须在键中分配新名称。例如，如果 Orders 表和 OrderDetails 表各自为它们的列使用不同的名称，则您将通过在匿名类型中分配相同的名称来创建组合键：

```
join...on new {Name = o.CustomerName, ID = o.CustID} equals
    new {Name = d.CustName, ID = d.CustID }
```

还可以在 **group** 子句中使用组合键。

编译代码

- 若要编译并运行此代码，请按下面的步骤进行操作：
- 打开如何：连接到 Northwind 数据库，并按照说明进行操作来设置项目并创建数据库连接。有关更多信息，请参见 如何：安装示例数据库。
- 在 samples.cs 中创建一个新的空方法，该方法采用名为 db 的 Northwind 输入参数（与该文件中的其他方法类似）。将此示例中的代码粘贴到方法体中。
- 修改 program.cs 以从 Main 中调用该新方法。
- 按 F5 编译并运行查询。

请参阅

LINQ 查询表达式（C# 编程指南）

join 子句（C# 参考）

group 子句（C# 参考）

如何：执行自定义联接操作（C# 编程指南）

此示例演示如何执行无法使用 **join** 子句执行的联接操作。在查询表达式中，**join** 子句仅适用于同等联接（这是迄今为止最常见的联接操作类型），并针对同等联接进行了优化。执行同等联接时，一般总是可以通过使用 **join** 子句获得最佳性能。

但是，在下面一些情况中，无法使用 **join** 子句：

- 联接是在不等式（非同等联接）上断言的。
- 联接是在多个等式或不等式上断言的。
- 必须在联接操作之前为右侧（内部）序列引入一个临时范围变量。

若要执行非同等联接，可以使用多个 **from** 子句单独引入每个数据源。然后，在 **where** 子句中将谓词表达式应用于每个源的范围变量。该表达式还可以采用方法调用的形式。

> 注意
>
> 不要将这种自定义联接操作与使用多个 **from** 子句访问内部集合相混淆。有关更多信息，请参见 join 子句（C# 参考）。

下面示例中的第一个方法演示了一个简单的交叉联接。必须慎用交叉联接，因为它们可能产生非常大的结果集。但在某些方案中，可以使用它们创建源序列以供运行附加查询。

第二个方法产生其类别 ID 列在左侧类别列表中的所有产品的序列。请注意，这种方法使用 **let** 子句和 **Contains** 方法创建了一个临时数组。还可以在查询前创建该数组并去掉第一个 **from** 子句。

```
class CustomJoins
{

    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
```

```
        List<Category> categories = new List<Category>()
 {
     new Category(){Name="Beverages", ID=001},
     new Category(){ Name="Condiments", ID=002},
     new Category(){ Name="Vegetables", ID=003},
 };

        // Specify the second data source.
        List<Product> products = new List<Product>()
{
   new Product{Name="Tea",  CategoryID=001},
   new Product{Name="Mustard", CategoryID=002},
   new Product{Name="Pickles", CategoryID=002},
   new Product{Name="Carrots", CategoryID=003},
   new Product{Name="Bok Choy", CategoryID=003},
   new Product{Name="Peaches", CategoryID=005},
   new Product{Name="Melons", CategoryID=005},
   new Product{Name="Ice Cream", CategoryID=007},
   new Product{Name="Mackerel", CategoryID=012},
 };
        #endregion

        static void Main()
        {
            CustomJoins app = new CustomJoins();
            app.CrossJoin();
            app.NonEquijoin();

            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }

        void CrossJoin()
        {
            var crossJoinQuery =
                from c in categories
                from p in products
                select new { c.ID, p.Name };

            Console.WriteLine("Cross Join Query:");
            foreach (var v in crossJoinQuery)
            {
                Console.WriteLine("{0,-5}{1}", v.ID, v.Name);
            }
        }

        void NonEquijoin()
        {
            var nonEquijoinQuery =
                from p in products
                let catIds = from c in categories
                             select c.ID
                where catIds.Contains(p.CategoryID) == true
```

```
                   select new { Product = p.Name, CategoryID = p.Cate

               Console.WriteLine("Non-equijoin query:");
               foreach (var v in nonEquijoinQuery)
               {
                   Console.WriteLine("{0,-5}{1}", v.CategoryID, v.Pro
               }
           }
       }
       /* Output:
   Cross Join Query:
   1    Tea
   1    Mustard
   1    Pickles
   1    Carrots
   1    Bok Choy
   1    Peaches
   1    Melons
   1    Ice Cream
   1    Mackerel
   2    Tea
   2    Mustard
   2    Pickles
   2    Carrots
   2    Bok Choy
   2    Peaches
   2    Melons
   2    Ice Cream
   2    Mackerel
   3    Tea
   3    Mustard
   3    Pickles
   3    Carrots
   3    Bok Choy
   3    Peaches
   3    Melons
   3    Ice Cream
   3    Mackerel
   Non-equijoin query:
   1    Tea
   2    Mustard
   2    Pickles
   3    Carrots
   3    Bok Choy
   Press any key to exit.
        */
```

在下面的示例中，查询必须基于匹配键联接两个序列，而对于内部（右侧）序列而言，无法在 join 子句本身之前获取这些键。如果此联接是使用 **join** 子句执行的，则必须为每个元素调用 **Split** 方法。使用多个 **from** 子句可使查询避免反复进行方

法调用的系统开销。然而，由于 **join** 进行了优化，因此在此特定情况下，它仍然可能比使用多个 **from** 子句快。结果会有所不同，主要取决于方法调用的系统开销有多大。

```
class MergeTwoCSVFiles
{
    static void Main()
    {
        // See section Compiling the Code for information about the
        string[] names = System.IO.File.ReadAllLines(@"../../../nar
        string[] scores = System.IO.File.ReadAllLines(@"../../../sc

        // Merge the data sources using a named type.
        // You could use var instead of an explicit type for the qu
        IEnumerable<Student> queryNamesScores =
            // Split each line in the data files into an array of s
            from name in names
            let x = name.Split(',')
            from score in scores
            let s = score.Split(',')
            // Look for matching IDs from the two data files.
            where x[2] == s[0]
            // If the IDs match, build a Student object.
            select new Student()
            {
                FirstName = x[0],
                LastName = x[1],
                ID = Convert.ToInt32(x[2]),
                ExamScores = (from scoreAsText in s.Skip(1)
                              select Convert.ToInt32(scoreAsText)).
                              ToList()
            };

        // Optional. Store the newly created student objects in mem
        // for faster access in future queries
        List<Student> students = queryNamesScores.ToList();

        foreach (var student in students)
        {
            Console.WriteLine("The average score of {0} {1} is {2}.
                student.FirstName, student.LastName, student.ExamSc
        }

        //Keep console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

class Student
{
    public string FirstName { get; set; }
```

```
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

/* Output:
    The average score of Omelchenko Svetlana is 82.5.
    The average score of O'Donnell Claire is 72.25.
    The average score of Mortensen Sven is 84.5.
    The average score of Garcia Cesar is 88.25.
    The average score of Garcia Debra is 67.
    The average score of Fakhouri Fadi is 92.25.
    The average score of Feng Hanying is 88.
    The average score of Garcia Hugo is 85.75.
    The average score of Tucker Lance is 81.75.
    The average score of Adams Terry is 85.25.
    The average score of Zabokritski Eugene is 83.
    The average score of Tucker Michael is 92.
 */
```

编译代码

- 创建一个面向 .NET Framework 3.5 或更高版本的 Visual Studio 控制台应用程序项目。默认情况下，该项目具有一个对 System.Core.dll 的引用以及一条针对 System.Linq 命名空间的 **using** 指令。
- 将 Program 类替换为上述示例中的代码。
- 按照 How to: Join Content from Dissimilar Files (LINQ) 中的指令设置数据文件（scores.csv 和 names.csv）。
- 按 F5 编译并运行程序。
- 按任意键退出控制台窗口。

请参阅

LINQ 查询表达式（C# 编程指南）

join 子句（C# 参考）

Join Operations

如何：对 Join 子句的结果进行排序（C# 编程指南）

如何：在查询表达式中处理 Null 值（C# 编程指南）

此示例演示如何处理源集合中可能的 null 值。诸如 IEnumerable<T> 等对象集合可能包含值为 null 的元素。如果源集合为 null 或包含值为 null 的元素，并且查询未处理 null 值，当您执行查询时将会引发 NullReferenceException。

您可以采用防御方式进行编码以避免 null 引用异常，如下面的示例中所示：

```
var query1 =
    from c in categories
    where c != null
    join p in products on c.ID equals
        (p == null ? null : p.CategoryID)
    select new { Category = c.Name, Name = p.Name };
```

在前面的示例中，**where** 子句筛选出类别序列中的所有 null 元素。此技术与 join 子句中的 null 检查无关。此示例中包含 null 的条件表达式将发挥作用，因为 Products.CategoryID 的类型为 **int?**（**Nullable<int>** 的简写形式）。

在 join 子句中，只要其中一个比较键是可以为 null 的类型，您就可以在查询表达式中将另一个比较键强制转换成可以为 null 的类型。在下面的示例中，假定 EmployeeID 是一个列，其中包含类型为 **int?** 的值：

```
void TestMethod(Northwind db)
{
    var query =
        from o in db.Orders
        join e in db.Employees
            on o.EmployeeID equals (int?)e.EmployeeID
        select new { o.OrderID, e.FirstName };
}
```

请参阅

Nullable<T>

LINQ 查询表达式（C# 编程指南）

可以为 null 的类型（C# 编程指南）

如何：在查询表达式中处理异常（C# 编程指南）

可以在查询表达式的上下文中调用任何方法。但是，建议您避免在查询表达式中调用任何可能产生副作用（例如，修改数据源的内容或引发异常）的方法。此示例演示在查询表达式中调用方法时如何避免引发异常，同时又不违反 .NET Framework 的有关异常处理的一般性准则。这些准则说明：如果您了解为什么在给定上下文中会引发特定异常，那么就可以捕捉该异常。有关更多信息，请参见异常的最佳做法。

最后一个示例演示如何处理那些必须在查询执行过程中引发异常的情况。

下面的示例演示如何将异常处理代码移至查询表达式外部。仅当该方法不依赖于查询的任何本地变量时，才能这样做。

```
class ExceptionsOutsideQuery
{
    static void Main()
    {
        // DO THIS with a datasource that might
        // throw an exception. It is easier to deal with
        // outside of the query expression.
        IEnumerable<int> dataSource;
        try
        {
            dataSource = GetData();
        }
        catch (InvalidOperationException)
        {
            // Handle (or don't handle) the exception
            // in the way that is appropriate for your application
            Console.WriteLine("Invalid operation");
            goto Exit;
        }

        // If we get here, it is safe to proceed.
        var query = from i in dataSource
                    select i * i;

        foreach (var i in query)
            Console.WriteLine(i.ToString());

        //Keep the console window open in debug mode
        Exit:
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // A data source that is very likely to throw an exception!
    static IEnumerable<int> GetData()
    {
        throw new InvalidOperationException();
    }
}
```

在某些情况下，对在查询内引发的异常的最佳响应可能是立即停止执行查询。下面的示例演示如何处理可能从查询正文内部引发的异常。假定 SomeMethodThatMightThrow 可能导致要求停止执行查询的异常。

请注意，**try** 块将 **foreach** 循环而不是查询本身封闭起来。这是因为 **foreach** 循环是实际执行查询的场所。有关更多信息，请参见 Introduction to LINQ Queries (C#)。

```
class QueryThatThrows
{
    static void Main()
    {
        // Data source.
        string[] files = { "fileA.txt", "fileB.txt", "fileC.txt" };

        // Demonstration query that throws.
        var exceptionDemoQuery =
            from file in files
            let n = SomeMethodThatMightThrow(file)
            select n;

        // Runtime exceptions are thrown when query is executed.
        // Therefore they must be handled in the foreach loop.
        try
        {
            foreach (var item in exceptionDemoQuery)
            {
                Console.WriteLine("Processing {0}", item);
            }
        }

        // Catch whatever exception you expect to raise
        // and/or do any necessary cleanup in a finally block
        catch (InvalidOperationException e)
        {
            Console.WriteLine(e.Message);
        }

        //Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Not very useful as a general purpose method.
    static string SomeMethodThatMightThrow(string s)
    {
        if (s[4] == 'C')
            throw new InvalidOperationException();
        return @"C:\newFolder\" + s;
    }
}
/* Output:
    Processing C:\newFolder\fileA.txt
    Processing C:\newFolder\fileB.txt
    Operation is not valid due to the current state of the object.
 */
```

编译代码

- 创建面向 .NET Framework 3.5 版的 Visual Studio 项目。默认情况下，该项目具有一个对 System.Core.dll 的引用以及一条针对 System.Linq 命名空间的 **using** 指令。
- 将代码复制到项目中。
- 按 F5 编译并运行程序。

按任意键退出控制台窗口。

请参阅

LINQ 查询表达式（C# 编程指南）

Main() 和命令行参数（C# 编程指南）

Main方法是 C# 控制台应用程序或窗口应用程序的入口点。（库和服务不要求将 **Main** 方法作为入口点。）应用程序启动时，**Main** 方法是第一个调用的方法。

C# 程序中只能有一个入口点。如果您有多个类都包含 **Main** 方法，则必须使用 **/main** 编译器选项编译您的程序，以指定用作入口点的 **Main** 方法。有关更多信息，请参见/main (C# Compiler Options)。

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments:
        System.Console.WriteLine(args.Length);
    }
}
```

概述

- **Main** 方法是 .exe 程序的入口点，程序控制流在该处开始和结束。
- **Main** 在类或结构内声明。 **Main** 必须是 静态，且不应该是 公开。（在前面的示例中，它接受默认访问级别 private。）但不要求封闭类或结构是静态的。
- **Main** 的返回类型有两种：**void** 或 **int**。
- 所声明的 **Main** 方法可以具有包含命令行实参的 string[] 形参，也可以不具有这样的形参。使用 Visual Studio 创建 Windows 窗体应用程序时，可以手动添加形参，也可以使用 Environment 类获取命令行实参。形参读取为零索引的命令行参数。与 C 和 C++ 不同，程序的名称视为第一个命令行参数。

本节内容

- 命令行参数（C# 编程指南）
- 如何：显示命令行参数（C# 编程指南）
- 如何：使用 foreach 访问命令行参数（C# 编程指南）
- Main() 返回值（C# 编程指南）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

Command-line Building With csc.exe

C# 编程指南

方法（C# 编程指南）

在 C# 程序内部

<paveover>C# Sample Applications

命令行参数（C# 编程指南）

通过以下方式之一定义方法，可以将参数发送至 **Main** 方法。

```
static int Main(string[] args)
```

```
static void Main(string[] args)
```

> 注意
>
> 若要在 Windows 窗体应用程序中的 **Main** 方法中启用命令行参数，必须手动修改 program.cs 中 **Main** 的签名。Windows 窗体设计器生成的代码创建没有输入参数的 **Main**。也可以使用 Environment.CommandLine 或 Environment.GetCommandLineArgs 从控制台或 Windows 应用程序中的任何位置访问命令行参数。

Main 方法的参数是表示命令行参数的 String 数组。一般是通过测试 Length 属性来确定参数是否存在，例如：

```
if (args.Length == 0)
{
    System.Console.WriteLine("Please enter a numeric argument.");
    return 1;
}
```

还可以使用 Convert 类或 **Parse** 方法将字符串参数转换为数值类型。例如，下面的语句使用 Parse 方法将 string 转换为 long 数字：

```
long num = Int64.Parse(args[0]);
```

也可以使用别名为 Int64 的 C# 类型 long：

```
long num = long.Parse(args[0]);
```

还可以使用 Convert 类的方法 ToInt64 完成同样的工作：

```
long num = Convert.ToInt64(s);
```

有关更多信息，请参见Parse和Convert。

下面的示例演示如何在控制台应用程序中使用命令行参数。应用程序在运行时采用一个参数，将该参数转换为整数，并计算该数的阶乘。如果没有提供参数，则应用程序发出一条消息来解释程序的正确用法。

若要根据命令提示编译并运行应用程序，请执行以下步骤：

1. 将以下代码粘贴到任何文本编辑器中，并将文件保存为名为 Factorial.cs 的文本文件。

```
//Add a using directive for System if the directive isn't alrea

public class Functions
{
    public static long Factorial(int n)
    {
        // Test for invalid input
        if ((n &lt; 0) || (n &gt; 20))
        {
            return -1;
        }

        // Calculate the factorial iteratively rather than recu
        long tempResult = 1;
        for (int i = 1; i &lt;= n; i++)
        {
            tempResult *= i;
        }
        return tempResult;
    }
}

class MainClass
{
    static int Main(string[] args)
    {
        // Test if input arguments were supplied:
        if (args.Length == 0)
        {
            System.Console.WriteLine("Please enter a numeric ar
            System.Console.WriteLine("Usage: Factorial &lt;num&
            return 1;
        }

        // Try to convert the input arguments to numbers. This
        // an exception if the argument is not a number.
        // num = int.Parse(args[0]);
        int num;
        bool test = int.TryParse(args[0], out num);
        if (test == false)
        {
```

```
                System.Console.WriteLine("Please enter a numeric ar
                System.Console.WriteLine("Usage: Factorial &lt;num&
                return 1;
            }

            // Calculate factorial.
            long result = Functions.Factorial(num);

            // Print result.
            if (result == -1)
                System.Console.WriteLine("Input must be &gt;= 0 and
            else
                System.Console.WriteLine("The Factorial of {0} is {

            return 0;
        }
    }
    // If 3 is entered on command line, the
    // output reads: The factorial of 3 is 6.
```

2. 从“开始”屏幕或“开始”菜单中，打开 Visual Studio“开发人员命令提示”窗口，然后导航到包含您刚创建的文件的文件夹。

3. 若要编译应用程序，请输入下面的命令。

   csc Factorial.cs

   如果您的应用程序中有没有编译错误，则将创建名为 Factorial.exe 的可执行文件。

4. 输入以下命令来计算 3 的阶乘：

   Factorial 3

5. 此命令将生成以下输出：The factorial of 3 is 6.

> 注意
>
> 在 Visual Studio 中运行应用程序时，可以在“项目设计器”->“调试”页中指定命令行参数。

有关如何使用命令行参数的更多示例，请参见如何：使用命令行创建和使用程序集（C# 和 Visual Basic）。

请参阅

System.Environment

C# 编程指南

Main() 和命令行参数（C# 编程指南）

如何：显示命令行参数（C# 编程指南）

如何：使用 foreach 访问命令行参数（C# 编程指南）

Main() 返回值（C# 编程指南）

类（C# 编程指南）

如何：显示命令行参数（C# 编程指南）

可以通过 Main 的可选参数来访问通过命令行提供给可执行文件的参数。参数以字符串数组的形式提供。数组的每个元素都包含一个参数。参数之间的空白被移除。例如，下面是对一个假想的可执行文件的命令行调用：

命令行输入	传递给 Main 的字符串数组
executable.exe a b c	"a""b"“c”
executable.exe one two	"one""two"
executable.exe "one two" three	"one two""three"

📝 注意
在 Visual Studio 中运行应用程序时，可以在“项目设计器”->“调试”页中指定命令行参数。

本示例显示了传递给命令行应用程序的命令行参数。显示的输出对应于上表中的第一项。

```
class CommandLine
{
    static void Main(string[] args)
    {
        // The Length property provides the number of array element
        System.Console.WriteLine("parameter count = {0}", args.Leng

        for (int i = 0; i < args.Length; i++)
        {
            System.Console.WriteLine("Arg[{0}] = [{1}]", i, args[i]
        }
    }
}
/* Output (assumes 3 cmd line args):
    parameter count = 3
    Arg[0] = [a]
    Arg[1] = [b]
    Arg[2] = [c]
*/
```

请参阅

C# 编程指南

Command-line Building With csc.exe

Main() 和命令行参数（C# 编程指南）

如何：使用 foreach 访问命令行参数（C# 编程指南）

Main() 返回值（C# 编程指南）

如何：使用 foreach 访问命令行参数（C# 编程指南）

循环访问数组的另一种方法是使用 foreach 语句，如下面的示例所示。 **foreach** 语句可以用于循环访问数组、.NET Framework 集合类或任何实现 IEnumerable 接口的类或结构。

> 注意
>
> 在 Visual Studio 中运行应用程序时，可以在“项目设计器”->“调试”页中指定命令行参数。

下面的示例演示如何使用 **foreach** 输出命令行参数。

```
// arguments: John Paul Mary
```

```
class CommandLine2
{
    static void Main(string[] args)
    {
        System.Console.WriteLine("Number of command line parameters

        foreach (string s in args)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
    Number of command line parameters = 3
    John
    Paul
    Mary
*/
```

请参阅

Array

System.Collections

Command-line Building With csc.exe

C# 编程指南

foreach，in（C# 参考）

Main() 和命令行参数（C# 编程指南）

如何：显示命令行参数（C# 编程指南）

Main() 返回值（C# 编程指南）

Main() 返回值（C# 编程指南）

Main 方法可以返回 **void**：

```
static void Main()
{
    //...
}
```

它还可以返回 **int**：

```
static int Main()
{
    //...
    return 0;
}
```

如果不使用 Main 的返回值，则返回 **void** 可以稍微简化代码。但是，如果返回整数，则程序可以与调用该可执行文件的其他程序或脚本交流状态信息。下面的示例演示如何访问 Main 的返回值。

此示例使用一个批处理文件来运行程序，并测试 Main 函数的返回值。在 Windows 中执行程序时，Main 函数返回的任何值都将存储在名为 ERRORLEVEL 的环境变量中。通过检查 ERRORLEVEL 变量，批处理文件可以确定执行结果。通常，返回值为零指示执行成功。下面是一个简单程序示例，从 Main 函数返回零。零指示程序成功运行。请将该程序保存为 MainReturnValTest.cs。

```
// Save this program as MainReturnValTest.cs.
class MainReturnValTest
{
    static int Main()
    {
        //...
        return 0;
    }
}
```

此示例使用了批处理文件，因此最好在命令提示中编译这段代码。请按照How to: Set Environment Variables for the Visual Studio Command Line中的说明启用命令行生成，或者使用 Visual Studio 命令提示（可通过“开始”菜单中的“**Visual Studio Tools**”访问）。在命令提示符下，定位到保存程序的文件夹。下面的命令编译 MainReturnValTest.cs，生成可执行文件 MainReturnValTest.exe。

csc MainReturnValTest.cs

接下来，创建一个批处理文件，运行 MainReturnValTest.exe 并显示结果。将下面的代码粘贴到文本文件中，将该文件另存为 test.bat，保存到包含 MainReturnValTest.cs 和 MainReturnValTest.exe 的文件夹中。在命令提示符下，键入 test，运行该批处理文件。

因为代码返回零，所以该批处理文件会报告成功。但是，如果将 MainReturnValTest.cs 更改为返回非零值，然后重新编译程序，则批处理文件的后续执行将报告失败。

```
rem test.bat
@echo off
MainReturnValTest
@if "%ERRORLEVEL%" == "0" goto good

:fail
    echo Execution Failed
    echo return value = %ERRORLEVEL%
    goto end

:good
    echo Execution succeeded
    echo Return value = %ERRORLEVEL%
    goto end

:end
```

示例输出

Execution succeeded

Return value = 0

请参阅

C# 编程指南

C# 参考

Main() 和命令行参数（C# 编程指南）

如何：显示命令行参数（C# 编程指南）

如何：使用 foreach 访问命令行参数（C# 编程指南）

命名空间（C# 编程指南）

使用 C# 编程时，通过两种方式来大量使用命名空间。首先，.NET Framework 使用命名空间来组织它的众多类，如下所示：

```
System.Console.WriteLine("Hello World!");
```

System 是一个命名空间，**Console** 是该命名空间中的类。可以使用 **using** 关键字，因此不必使用完整的名称，如以下示例所示：

```
using System;
```

```
Console.WriteLine("Hello");
Console.WriteLine("World!");
```

有关更多信息，请参见 using 指令（C# 参考）。

其次，在较大的编程项目中，声明自己的命名空间可以帮助控制类名称和方法名称的范围。使用 namespace 关键字可声明命名空间，如下例所示：

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

命名空间概述

命名空间具有以下属性：

- 组织大型代码项目。
- 使用 **.** 运算符将它们分隔。
- **using directive** 不必为每个类指定命名空间的名称。

- **global** 命名空间是"根"命名空间：**global::System** 始终引用 .NET Framework 命名空间 **System**。

相关章节

有关命名空间的更多信息，请参见下列主题：

- 使用命名空间（C# 编程指南）
- 如何：使用全局命名空间别名（C# 编程指南）
- 如何：使用 My 命名空间（C# 编程指南）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

命名空间关键字（C# 参考）

using 指令（C# 参考）

:: 运算符（C# 参考）

. 运算符（C# 参考）

使用命名空间（C# 编程指南）

在 C# 程序中，通过两种方式来大量使用命名空间。首先，.NET Framework 类使用命名空间来组织它的众多类。其次，在较大的编程项目中，声明自己的命名空间可以帮助控制类和方法名的范围。

访问命名空间

大多数 C# 应用程序从一个 **using** 指令节开始。该节列出应用程序将会频繁使用的命名空间，避免程序员在每次使用其中包含的方法时都要指定完全限定的名称。

例如，通过在程序开头包括行：

```
using System;
```

程序员可以使用代码：

```
Console.WriteLine("Hello, World!");
```

而不是：

```
System.Console.WriteLine("Hello, World!");
```

命名空间别名

using 指令（C# 参考）还可用于创建命名空间的别名。例如，如果使用包含嵌套命名空间的以前编写的命名空间，您可能希望声明一个别名来提供引用特定命名空间的简写方法，如以下示例中所示：

```
using Co = Company.Proj.Nested;  // define an alias to represent a
```

使用命名空间来控制范围

namespace 关键字用于声明一个范围。在项目中创建范围的能力有助于组织代码，并可让您创建全局唯一的类型。在下面的示例中，名为 SampleClass 的类在两个命名空间中定义，其中一个命名空间嵌套在另一个之内。 . 运算符（C# 参考）用于区分所调用的方法。

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }

    // Create a nested namespace, and define another class.
    namespace NestedNamespace
    {
        class SampleClass
        {
            public void SampleMethod()
            {
                System.Console.WriteLine(
                  "SampleMethod inside NestedNamespace");
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Displays "SampleMethod inside SampleNamespace."
            SampleClass outer = new SampleClass();
            outer.SampleMethod();

            // Displays "SampleMethod inside SampleNamespace."
            SampleNamespace.SampleClass outer2 = new SampleNamespac
            outer2.SampleMethod();

            // Displays "SampleMethod inside NestedNamespace."
            NestedNamespace.SampleClass inner = new NestedNamespace
            inner.SampleMethod();
        }
    }
}
```

完全限定名

命名空间和类型的名称必须唯一，由指示逻辑层次结构的完全限定名描述。例如，语句 A.B 表示 A 是命名空间或类型的名称，而 B 则嵌套在其中。

下面的示例中有嵌套的类和命名空间。在每个实体的后面，需要完全限定名作为注释。

```
namespace N1     // N1
{
    class C1      // N1.C1
    {
        class C2   // N1.C1.C2
        {
        }
    }
    namespace N2  // N1.N2
    {
        class C2   // N1.N2.C2
        {
        }
    }
}
```

在以上代码段中：

- 命名空间 N1 是全局命名空间的成员。它的完全限定名是 N1。
- 命名空间 N2 是命名空间 N1 的成员。它的完全限定名是 N1.N2。
- 类 C1 是 N1 的成员。它的完全限定名是 N1.C1。
- 在此代码中使用了两次 C2 类名。但是，完全限定名是唯一的。 C2 的第一个实例是在 C1 中声明的；因此，其完全限定名为：N1.C1.C2。 C2 的第二个实例是在命名空间 N2 中声明的；因此，其完全限定名为：N1.N2.C2。

使用以上代码段，可以用以下方法将新的类成员 C3 添加到命名空间 N1.N2 内：

```
namespace N1.N2
{
    class C3   // N1.N2.C3
    {
    }
}
```

一般情况下，应使用 :: 来引用命名空间别名或使用 **global::** 来引用全局命名空间，并使用 **.** 来限定类型或成员。

与引用类型而不是命名空间的别名一起使用 :: 是错误的。例如：

```
using Alias = System.Console;
```

```
  class TestClass
{
    static void Main()
    {
        // Error
        //Alias::WriteLine("Hi");

        // OK
        Alias.WriteLine("Hi");
    }
}
```

记住单词 **global** 不是预定义的别名，因此 global.X 没有任何特殊的含义。仅当与 :: 一起使用时，它才获得特殊的含义。

定义名为 global 的别名会生成编译器警告 CS0440，因为 **global::** 始终引用全局命名空间而不是别名。例如，下面的行将产生警告：

```
using global = System.Collections;   // Warning
```

最好将 :: 与别名一起使用，这样可以避免意外引入其他类型。以下面的代码为例：

```
  using Alias = System;
```

```
  namespace Library
{
    public class C : Alias.Exception { }
}
```

这样做可行，但是如果接着引入一个名为 Alias 的类型，则 Alias. 将改为绑定到该类型。使用 Alias::Exception 可以确保 Alias 被当作命名空间别名，而不会被误认为类型。

有关 **global** 别名的更多信息，请参见主题如何：使用全局命名空间别名（C# 编程指南）。

请参阅

C# 编程指南

命名空间（C# 编程指南）

命名空间关键字（C# 参考）

. 运算符（C# 参考）

:: 运算符（C# 参考）

extern（C# 参考）

如何：使用全局命名空间别名（C# 编程指南）

当成员可能被同名的其他实体隐藏时，能够访问全局命名空间中的成员非常有用。

例如，在下面的代码中，Console 在 System 命名空间中解析为 TestApp.Console 而不是 **Console** 类型。

```
using System;
```

```
class TestApp
{
    // Define a new class called 'System' to cause problems.
    public class System { }

    // Define a constant called 'Console' to cause more problems.
    const int Console = 7;
    const int number = 66;

    static void Main()
    {
        // The following line causes an error. It accesses TestApp
        // which is a constant.
        //Console.WriteLine(number);
    }
}
```

由于类 TestApp.System 隐藏了 **System** 命名空间，因此使用 **System.Console** 仍然会导致错误：

```
// The following line causes an error. It accesses TestApp.System,
// which does not have a Console.WriteLine method.
System.Console.WriteLine(number);
```

但是，可以通过使用 global::System.Console 避免这一错误，如下所示：

```
// OK
global::System.Console.WriteLine(number);
```

当左侧的标识符为 **global** 时，对右侧标识符的搜索将从全局命名空间开始。例如，下面的声明将 TestApp 作为全局空间的一个成员进行引用。

```
class TestClass : global::TestApp
```

显然，并不推荐创建自己的名为 System 的命名空间，您不可能遇到出现此情况的任何代码。但是，在较大的项目中，很有可能在一个窗体或其他窗体中出现命名空间重复。在这种情况下，全局命名空间限定符可保证您可以指定根命名空间。

在此示例中，命名空间 **System** 用于包括类 TestClass，因此必须使用 global::System.Console 来引用 **System.Console** 类，该类被 **System** 命名空间隐藏。而且，别名 colAlias 用于引用命名空间 **System.Collections**；因此，将使用此别名而不是命名空间来创建 System.Collections.Hashtable 的实例。

```
using colAlias = System.Collections;
namespace System
{
    class TestClass
    {
        static void Main()
        {
            // Searching the alias:
            colAlias::Hashtable test = new colAlias::Hashtable();

            // Add items to the table.
            test.Add("A", "1");
            test.Add("B", "2");
            test.Add("C", "3");

            foreach (string name in test.Keys)
            {
                // Searching the global namespace:
                global::System.Console.WriteLine(name + " " + test
            }
        }
    }
}
```

```
A 1
B 2
C 3
```

请参阅

C# 编程指南

命名空间（C# 编程指南）

. 运算符（C# 参考）

:: 运算符（C# 参考）

extern（C# 参考）

如何：使用 My 命名空间（C# 编程指南）

Microsoft.VisualBasic.MyServices 命名空间（Visual Basic 中的 **My**）提供对许多 .NET Framework 类的简单直观的访问，使您能够编写可与计算机、应用程序、设置、资源等交互的代码。虽然 **MyServices** 命名空间最初是为使用 Visual Basic 而设计的，但它也可以在 C# 应用程序中使用。

有关在 Visual Basic 中使用 **MyServices** 命名空间的更多信息，请参见 使用 My 开发 (Visual Basic)。

添加引用

在解决方案中使用 **MyServices** 类之前，必须添加一个对 Visual Basic 库的引用。

添加对 **Visual Basic** 库的引用

1. 在“解决方案资源管理器”中右击“引用”节点，再选择“添加引用”。
2. 出现“引用”对话框后，向下滚动列表，选择“Microsoft.VisualBasic.dll”。

   您可能还需要在程序开头的 **using** 节中包括以下行。

   ```
   using Microsoft.VisualBasic.Devices;
   ```

此示例调用 **MyServices** 命名空间中包含的各种静态方法。要编译此代码，必须在项目中添加一个对 Microsoft.VisualBasic.DLL 的引用。

```
using System;
using Microsoft.VisualBasic.Devices;

class TestMyServices
{
    static void Main()
    {
        // Play a sound with the Audio class:
        Audio myAudio = new Audio();
        Console.WriteLine("Playing sound...");
        myAudio.Play(@"c:\WINDOWS\Media\chimes.wav");

        // Display time information with the Clock class:
        Clock myClock = new Clock();
        Console.Write("Current day of the week: ");
        Console.WriteLine(myClock.LocalTime.DayOfWeek);
        Console.Write("Current date and time: ");
        Console.WriteLine(myClock.LocalTime);

        // Display machine information with the Computer class:
        Computer myComputer = new Computer();
        Console.WriteLine("Computer name: " + myComputer.Name);

        if (myComputer.Network.IsAvailable)
        {
            Console.WriteLine("Computer is connected to network.");
        }
        else
        {
            Console.WriteLine("Computer is not connected to networl
        }
    }
}
```

并不是 **MyServices** 命名空间中的所有的类都可以从 C# 应用程序调用：例如 FileSystemProxy 类就不兼容。在这种特定情况下，可以改用作为 FileSystem（它也包含在 VisualBasic.dll 中）的一部分的静态方法。例如，下面介绍了如何使用这样的方法来复制目录：

```
// Duplicate a directory
Microsoft.VisualBasic.FileIO.FileSystem.CopyDirectory(
    @"C:\original_directory",
    @"C:\copy_of_original_directory");
```

请参阅

C# 编程指南

命名空间（C# 编程指南）

使用命名空间（C# 编程指南）

可以为 null 的类型（C# 编程指南）

可以为 null 的类型是 System.Nullable<T> 结构的实例。可以为 null 的类型可以表示其基础值类型正常范围内的值，再加上一个 **null** 值。例如，Nullable<Int32> 读作“可以为 null 的 Int32”，可以将其赋值为 -2147483648 到 2147483647 之间的任意值，也可以将其赋值为 **null** 值。可以赋给 Nullable<bool> 的值包括 true、false 或 null。在处理数据库和其他包含不可赋值的元素的数据类型时，将 **null** 赋值给数值类型或布尔型的功能特别有用。例如，数据库中的布尔型字段可以存储值 **true** 或 **false**，或者，该字段也可以未定义。

```
class NullableExample
{
    static void Main()
    {
        int? num = null;

        // Is the HasValue property true?
        if (num.HasValue)
        {
            System.Console.WriteLine("num = " + num.Value);
        }
        else
        {
            System.Console.WriteLine("num = Null");
        }

        // y is set to zero
        int y = num.GetValueOrDefault();

        // num.Value throws an InvalidOperationException if num.Has
        try
        {
            y = num.Value;
        }
        catch (System.InvalidOperationException e)
        {
            System.Console.WriteLine(e.Message);
        }
    }
}
```

此示例将显示输出：

num = Null

Nullable object must have a value.

有关更多示例，请参见使用可以为 null 的类型（C# 编程指南）。

可以为 null 的类型概述

可以为 null 的类型具有以下特性：

- 可以为 null 的类型表示可被赋值为 **null** 值的值类型变量。无法创建基于引用类型的可以为 null 的类型。（引用类型已支持 **null** 值。）
- 语法 **T?** 是 Nullable<T> 的简写，此处的 **T** 为值类型。这两种形式可以互换。
- 为可以为 null 的类型赋值的方法与为一般值类型赋值的方法相同，如 int? x = 10; 或 double? d = 4.108。对于可以为 null 的类型，也可向其赋 **null**: int? x = null. 值
- 如果基础类型的值为 **null**，请使用 Nullable<T>.GetValueOrDefault 方法返回该基础类型所赋的值或默认值，例如 int j = x.GetValueOrDefault();
- 将 HasValue 和 Value 只读属性用于测试是否为空和检索值，如下面的示例所示：if(x.HasValue) j = x.Value;
    - 如果此变量包含值，则 **HasValue** 属性返回 **true**；或者如果是 **null** 则返回 **false**。
    - 如果已赋值，则 **Value** 属性返回该值。否则，将引发 System.InvalidOperationException。
    - **HasValue** 的默认值为 **false**。 **Value** 属性没有默认值。
    - 还可以将 **==** 和 **!=** 操作数用于可为 null 的类型，如下面的示例所示：if (x != null) y = x;
- 使用 **??** 算符分配默认值，在将当前值为 **null** 的可以为 null 的类型赋值给不可以为 null 的类型时，将应用该默认值，如 int? x = null; int y = x ?? -1;
- 不允许使用嵌套的可以为 null 的类型。将不编译下面一行：Nullable<Nullable<int>> n;

相关章节

有关更多信息：

- 使用可以为 null 的类型（C# 编程指南）
- 装箱可以为 null 的类型（C# 编程指南）
- ?? 运算符（C# 参考）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

Nullable

C# 编程指南

Visual C#

C# 参考

“lifted 正确意味着什么？

使用可以为 null 的类型（C# 编程指南）

可以为 null 的类型可以表示基础类型的所有值，另外还可以表示 null 值。可以为 null 的类型可通过下面两种方式中的一种声明：

System.Nullable<T> variable

- 或 -

T?variable

T 是可以为 null 的类型的基础类型。 T 可以是包括 **struct** 在内的任何值类型；但不能是引用类型。

有关可能使用可以为 null 的类型的示例，请考虑普通的布尔变量如何能够具有两个值：true 和 false。不存在表示“未定义”的值。在很多编程应用中（最突出的是数据库交互），变量可以以未定义的状态出现。例如，数据库中的某个字段可能包含值 true 或 false，但是它也可能根本不包含值。同样，可以将引用类型设置为 **null**，以指示它们未初始化。

这种不一致会导致额外的编程工作，如使用附加变量来存储状态信息、使用特殊值，等等。可以为 null 的类型修饰符使 C# 能够创建表示未定义值的值类型变量。

可以为 null 的类型示例

任何值类型都可用作可以为 null 的类型的基础。例如：

```
int? i = 10;
double? d1 = 3.14;
bool? flag = null;
char? letter = 'a';
int?[] arr = new int?[10];
```

可以为 null 的类型的成员

可以为 null 的类型的每个实例都具有两个公共的只读属性：

- **HasValue**

  HasValue 属于 **bool** 类型。当变量包含非 null 值时，它被设置为 **true**。

- **Value**

  Value 的类型与基础类型相同。如果 HasValue 为 **true**，则说明 Value 包含有意义的值。如果 HasValue 为 **false**，则访问 Value 将引发 InvalidOperationException。

在此示例中，HasValue 成员用于在尝试显示变量之前测试它是否包含值。

```
int? x = 10;
if (x.HasValue)
{
    System.Console.WriteLine(x.Value);
}
else
{
    System.Console.WriteLine("Undefined");
}
```

也可以如下面的示例所示对值进行测试：

```
int? y = 10;
if (y != null)
{
    System.Console.WriteLine(y.Value);
}
else
{
    System.Console.WriteLine("Undefined");
}
```

显式转换

可以为 null 的类型可强制转换为常规类型，方法是使用强制转换来显式转换或者通过使用 Value 属性来转换。例如：

```
int? n = null;

//int m1 = n;      // Will not compile.
int m2 = (int)n;   // Compiles, but will create an exception if n i
int m3 = n.Value;  // Compiles, but will create an exception if n i
```

如果两种数据类型之间定义了用户定义的转换，则同一转换也可用于这些数据类型的可以为 null 的版本。

隐式转换

可使用 **null** 关键字将可以为 null 的类型的变量设置为 null，如以下示例所示：

```
int? n1 = null;
```

从普通类型到可以为 null 的类型的转换是隐式的。

```
int? n2;
n2 = 10;  // Implicit conversion.
```

运算符

可以为 null 的类型还可以使用预定义的一元和二元运算符，以及现有的任何用户定义的值类型运算符。如果操作数为 null，这些运算符将产生一个 null 值；否则运算符将使用包含的值来计算结果。例如：

```
int? a = 10;
int? b = null;

a++;         // Increment by 1, now a is 11.
a = a * 10;  // Multiply by 10, now a is 110.
a = a + b;   // Add b, now a is null.
```

在对可以为 null 的类型执行比较时，如果其中一个可以为 null 的类型的值为 null，但另外一个类型的值不为 null，则除 **!=**（不等于）外，所有比较的结果都将为 **false**。一定不要以为由于一个特定比较的结果为 **false**，相反的情况就会为 **true**。在以下示例中，10 不大于、小于或等于 null。只有 num1 != num2 的计算结果为 **true**。

```
int? num1 = 10;
int? num2 = null;
if (num1 >= num2)
{
    Console.WriteLine("num1 is greater than or equal to num2");
}
else
{
    // This clause is selected, but num1 is not less than num2.
    Console.WriteLine("num1 >= num2 returned false (but num1 < num2
}

if (num1 < num2)
{
    Console.WriteLine("num1 is less than num2");
}
else
{
    // The else clause is selected again, but num1 is not greater t
    // or equal to num2.
    Console.WriteLine("num1 < num2 returned false (but num1 >= num2
}

if (num1 != num2)
{
    // This comparison is true, num1 and num2 are not equal.
    Console.WriteLine("Finally, num1 != num2 returns true!");
}

// Change the value of num1, so that both num1 and num2 are null.
num1 = null;
if (num1 == num2)
{
    // The equality comparison returns true when both operands are
    Console.WriteLine("num1 == num2 returns true when the value of
}

/* Output:
 * num1 >= num2 returned false (but num1 < num2 also is false)
 * num1 < num2 returned false (but num1 >= num2 also is false)
 * Finally, num1 != num2 returns true!
 * num1 == num2 returns true when the value of each is null
 */
```

如果两个可以为 null 的类型的值均为 null，则其相等比较的计算结果为 **true**。

??运算符

?? 运算符定义在将可以为 null 的类型分配给非可以为 null 的类型时返回的默认值。

```
int? c = null;

// d = c, unless c is null, in which case d = -1.
int d = c ?? -1;
```

此运算符还可用于多个可以为 null 的类型。例如：

```
int? e = null;
int? f = null;

// g = e or f, unless e and f are both null, in which case g = -1.
int g = e ?? f ?? -1;
```

bool?type

bool? 可以为 null 的类型可以包含三个不同的值：true、false 和 null。有关如何从 bool? 强制转换为 bool 的信息，请参见如何：安全地将 bool? 强制转换为 bool（C# 编程指南）。

可以为 null 的布尔值类似于 SQL 中使用的布尔变量类型。若要确保由 **&** 和 产生的结果**|** 运算符与 SQL 中的三值布尔类型一致，提供了以下预定义的运算符：

bool?operator &(bool?x, bool?y)

bool?operator |(bool?x, bool?y)

下表中列出了这些运算符的结果：

X	y	x&y	x|y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

请参阅

C# 编程指南

可以为 null 的类型（C# 编程指南）

装箱可以为 null 的类型（C# 编程指南）

可以为 Null 的值类型 (Visual Basic)

装箱可以为 null 的类型（C# 编程指南）

基于可以为 null 的类型的对象只在该对象为非空时装箱。如果 HasValue 为 **false**，则将对象引用赋值为 **null**，而不进行装箱。例如：

```
bool? b = null;
object o = b;
// Now o is null.
```

如果对象非空，也就是说，如果 HasValue 为 **true**，则会发生装箱过程，但只将可以为 null 的对象所基于的基础类型装箱。如果将非空的可以为 null 的值类型装箱，将使值类型本身（而不是包装该值类型的 System.Nullable<T>）装箱。例如：

```
bool? b = false;
int? i = 44;
object bBoxed = b; // bBoxed contains a boxed bool.
object iBoxed = i; // iBoxed contains a boxed int.
```

对于那些通过装箱非可以为 null 的类型而创建的类型来说，两种装箱对象是完全相同的。并且，像非可以为 null 的装箱类型一样，可以将它们取消装箱，使其成为可以为 null 的类型，如以下示例所示：

```
bool? b2 = (bool?)bBoxed;
int? i2 = (int?)iBoxed;
```

备注

可以为 null 的类型在装箱时的行为具有两个优点：

1. 可以测试可以为 null 的对象及其装箱的对应项是否为空：

```
bool? b = null;
object boxedB = b;
if (b == null)
{
  // True.
}
if (boxedB == null)
{
  // Also true.
}
```

2. 装箱的可以为 null 的类型完全支持基础类型的功能：

```
double? d = 44.4;
object iBoxed = d;
// Access IConvertible interface implemented by double.
IConvertible ic = (IConvertible)iBoxed;
int i = ic.ToInt32(null);
string str = ic.ToString();
```

请参阅

C# 编程指南

可以为 null 的类型（C# 编程指南）

如何：标识可以为 null 的类型（C# 编程指南）

如何：标识可以为 null 的类型（C# 编程指南）

可以使用 C# typeof 运算符来创建表示可以为 null 的类型的 Type 对象：

```
System.Type type = typeof(int?);
```

还可以使用 System.Reflection 命名空间的类和方法来生成表示可以为 null 的类型的 Type 对象。但是，如果您尝试使用 GetType 方法或 **is** 运算符在运行时获得可以为 null 的类型变量的类型信息，得到的结果是表示基础类型而不是可以为 null 的类型本身的 Type 对象。

如果对可以为 null 的类型调用 **GetType**，则在该类型被隐式转换为 Object 时将执行装箱操作。因此，GetType 总是返回表示基础类型而不是可以为 null 的类型的 Type 对象。

```
int? i = 5;
Type t = i.GetType();
Console.WriteLine(t.FullName); //"System.Int32"
```

C# 的 is 运算符还可以作用于可以为 null 的的基础类型。因此，不能使用 **is** 来确定变量是否为可以为 null 的类型。下面的示例演示 **is** 运算符将 Nullable<int> 变量视为 int 变量。

```
static void Main(string[] args)
{
  int? i = 5;
  if (i is int) // true
    //…
}
```

使用下面的代码来确定 Type 对象是否表示可以为 null 的类型。请记住，如果 **Type** 对象是通过调用 GetType 返回的，则此代码始终返回 false，如本主题中先前所述。

```
if (type.IsGenericType && type.GetGenericTypeDefinition() == typeof
```

请参阅

可以为 null 的类型（C# 编程指南）

装箱可以为 null 的类型（C# 编程指南）

如何：安全地将 bool? 强制转换为 bool（C# 编程指南）

bool? 可以为 null 的类型可以包含三个不同的值：**true**、**false** 和 **null**。因此，**bool?** 类型不能用于条件语句，如 **if**、**for** 或 **while**。例如，以下代码会导致编译器错误。

```
bool? b = null;
if (b) // Error CS0266.
{
}
```

这是不允许的，因为 **null** 在条件上下文中的含义并不清楚。若要在条件语句中使用 **bool?**，请首先检查其 HasValue 属性以确保其值不是 **null**，然后将它强制转换为 **bool**。有关更多信息，请参见 bool。如果对使用 **null** 值的 **bool?** 执行强制转换，则在条件测试中将引发 InvalidOperationException。下面的示例演示了一种从 **bool?** 安全地强制转换为 **bool** 的方法：

```
bool? test = null;
// Other code that may or may not
// give a value to test.
if(!test.HasValue) //check for a value
{
    // Assume that IsInitialized
    // returns either true or false.
    test = IsInitialized();
}
if((bool)test) //now this cast is safe
{
   // Do something.
}
```

请参阅

C# 编程指南

文字关键字（C# 参考）

可以为 null 的类型（C# 编程指南）

?? 运算符（C# 参考）

语句、表达式和运算符（C# 编程指南）

构成应用程序的 C# 代码由包含关键字、表达式和运算符的语句组成。本节包含关于 C# 程序的基本元素的信息。

有关更多信息，请参见：

- 语句（C# 编程指南）
- 表达式（C# 编程指南）
- 运算符（C# 编程指南）
- 匿名函数（C# 编程指南）
- 可重载运算符（C# 编程指南）
- 转换运算符（C# 编程指南）
    - 使用转换运算符（C# 编程指南）
    - 如何：在结构间实现用户定义的转换（C# 编程指南）
- 如何：使用运算符重载创建复数类（C# 编程指南）
- 相等比较（C# 编程指南）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

强制转换和类型转换（C# 编程指南）

语句（C# 编程指南）

程序所执行的操作以“语句”表达。常见操作包括声明变量、赋值、调用方法、循环访问集合，以及根据给定条件分支到一个或另一个代码块。语句在程序中的执行顺序称为“控制流”或“执行流”。根据程序对运行时所收到的输入的响应，在程序每次运行时控制流可能有所不同。

语句可以是以分号结尾的单行代码，或者是语句块中的一系列单行语句。语句块括在括号 {} 中，并且可以包含嵌套块。下面的代码演示两个单行语句示例和一个多行语句块：

```
    static void Main()
    {
        // Declaration statement.
        int counter;

        // Assignment statement.
        counter = 1;

        // Error! This is an expression, not an expression statemen
        // counter + 1;

        // Declaration statements with initializers are functionall
        // equivalent to  declaration statement followed by assignm
        int[] radii = { 15, 32, 108, 74, 9 }; // Declare and initia
        const double pi = 3.14159; // Declare and initialize  const

        // foreach statement block that contains multiple statement
        foreach (int radius in radii)
        {
            // Declaration statement with initializer.
            double circumference = pi * (2 * radius);

            // Expression statement (method invocation). A single-l
            // statement can span multiple text lines because line
            // are treated as white space, which is ignored by the
            System.Console.WriteLine("Radius of circle #{0} is {1}.
                                    counter, radius, circumference)

            // Expression statement (postfix increment).
            counter++;

        } // End of foreach statement block
    } // End of Main method body.
} // End of SimpleStatements class.
/*
   Output:
    Radius of circle #1 = 15\. Circumference = 94.25
    Radius of circle #2 = 32\. Circumference = 201.06
    Radius of circle #3 = 108\. Circumference = 678.58
    Radius of circle #4 = 74\. Circumference = 464.96
    Radius of circle #5 = 9\. Circumference = 56.55
*/
```

语句的类型

下表列出 C# 中的各种语句类型及其关联的关键字，并提供指向包含更多信息的主题的链接：

类别	C# 关键字/说明
声明语句	声明语句引入新的变量或常量。变量声明可以选择为变量赋值。在常量声明中必须赋值。

```
// Variable declaration statements.
double area;
double radius = 2;

// Constant declaration statement.
const double pi = 3.14159;
```

表达式语句	用于计算值的表达式语句必须在变量中存储该值。

```
// Expression statement (assignment).
area = 3.14 * (radius * radius);

// Error. Not  statement because no assignment:
//circ * 2;

// Expression statement (method invocation).
System.Console.WriteLine();

// Expression statement (new object creation).
System.Collections.Generic.List&lt;string&gt; strings =
    new System.Collections.Generic.List&lt;string&gt;();
```

选择语句	选择语句用于根据一个或多个指定条件分支到不同的代码段。有关更多信息，请参见下列主题：if, else, switch, case
迭代语句	迭代语句用于遍历集合（如数组），或重复执行同一组语句直到满足指定的条件。有关更多信息，请参见下列主题：do, for, foreach, in, while
跳转语句	跳转语句将控制转移给另一代码段。有关更多信息，请参见下列主题：break, continue, default, goto, return，yield
异常处理语句	异常处理语句用于从运行时发生的异常情况正常恢复。有关更多信息，请参见下列主题：throw, try-catch, try-finally, try-catch-finally
检查和未检查	检查和未检查语句用于指定当将结果存储在变量中、但该变量过小而不能容纳结果值时，是否允许数值运算导致溢出。有关更多信息，请参见检查和未检查。
await 语句	如果标记与 异步 修饰符的方法，在方法可以使用 等待 运算符。当控件移到在异步方法中的一个 **await** 表达式，控件回调用方，因此，在方法的进度挂起，直到等待任务完成。当任务完成后，执行在方法可以恢复。有关简单示例，请参见 方法（C# 编程指南）"Async "方法"一节。有关更多信息，请参见使用 Async 和 Await 的异步编程（C# 和 Visual Basic）。
yield return 语句	迭代器对集合的自定义迭代，如列表或数组。迭代器使用 将返回 语句返回每个元素一个节点。当 **yield return** 语句时，代码的当前位置确保。迭代器，当下次时，调用执行从该位置进行重新启动。有关更多信息，请参见迭代器（C# 和 Visual Basic）。
fixed 语句	Fixed 语句禁止垃圾回收器重定位可移动的变量。有关更多信息，请参见 fixed。
lock 语句	lock 语句用于限制一次仅允许一个线程访问代码块。有关更多信息，请参见 lock。
标记语句	可以为语句指定一个标记，然后使用 goto 关键字跳转到该标记语句。（参见下一行中的示例。）
空语句	空语句只含一个分号。空语句不执行任何操作，可以在需要语句但不需要执行任何操作的地方使用。下面的示例演示空语句的两种用法：

```
void ProcessMessages()
{
    while (ProcessMessage())
        ; // Statement needed here.
}

void F()
{
    //...
    if (done) goto exit;
//...
exit:
    ; // Statement needed here.
}
```

嵌入语句

一些语句（例如 do、while、for 和 foreach）后面始终跟有一条嵌入语句。此嵌入语句可以是单个语句，也可以是语句块中括在括号 {} 内的多个语句。甚至可以在括号 {} 内包含单行嵌入语句，如下面的示例所示：

```
// Recommended style. Embedded statement in  block.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
{
    System.Console.WriteLine(s);
}

// Not recommended.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
    System.Console.WriteLine(s);
```

未括在括号 {} 内的嵌入语句不能作为声明语句或标记语句。下面的示例演示了这种情况：

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

将该嵌入语句放在语句块中以修复错误：

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToLongDateString());
}
```

嵌套语句块

语句块可以嵌套，如以下代码所示：

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }

}
return "Not found.";
```

无法访问的语句

如果编译器认为在任何情况下控制流都无法到达特定语句，将生成警告 CS0162，如下面的示例所示：

```
// An over-simplified example of unreachable code.
const int val = 5;
if (val < 4)
{
    System.Console.WriteLine("I'll never write anything."); //CS016
}
```

相关章节

- 语句关键字（C# 参考）
- 表达式（C# 编程指南）

- 运算符（C# 编程指南）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

表达式（C# 编程指南）

“表达式”是由一个或多个操作数以及零个或零个以上的运算符所组成的序列，可以通过计算得到一个值、对象、方法或命名空间等结果。表达式可以包含文本值、方法调用、运算符及其操作数，或简单名称。简单名称可以是变量、类型成员、方法参数、命名空间或类型的名称。

表达式可以使用运算符，而运算符又可以将其他表达式用作参数，或者使用方法调用，而方法调用的参数又可以是其他方法调用，因此表达式既可以非常简单，也可以非常复杂。下面是表达式的两个示例：

```
((x < 10) && ( x > 5)) || ((x > 20) && (x < 25))
System.Convert.ToInt32("35")
```

表达式值

在大部分使用表达式的上下文中，例如在语句或方法参数中，表达式应计算为某个值。如果 x 和 y 是整数，表达式 x + y 将计算为一个数值。表达式 new MyClass() 计算为对 MyClass 对象的新实例的引用。表达式 myClass.ToString() 计算为一个字符串，因为字符串是该方法的返回类型。然而，虽然命名空间名称归类为表达式，但它不计算为值，因此永远不能作为任何表达式的最终结果。命名空间名称不能传递给方法参数，不能用在新表达式中，也不能赋值给变量。命名空间名称只能用作较大表达式的子表达式。同样如此的还有类型（与 System.Type 对象不同）、方法组名称（与特定方法不同）以及事件 add 和 remove 访问器。

每个值都有关联的类型。例如，如果 x 和 y 都是 **int** 类型的变量，则表达式 x + y 的值的类型也是 **int**。如果将该值赋给不同类型的变量，或者如果 x 和 y 是不同的类型，则应用类型转换规则。有关如何进行这种转换的更多信息，请参见强制转换和类型转换（C# 编程指南）。

溢出

如果值大于值类型的最大值，数值表达式可能导致溢出。有关更多信息，请参见 Checked 和 Unchecked（C# 参考）和 显式数值转换表（C# 参考）。

运算符的优先级和结合性

计算表达式的方式由结合性和运算符优先级控制。有关更多信息，请参见 运算符（C# 编程指南）。

除赋值表达式和方法调用表达式之外，大部分表达式都必须嵌在语句中。有关更多信息，请参见 语句（C# 编程指南）。

文本和简单名称

最简单的两种表达式类型是文本和简单名称。文本是没有名称的常数值。例如，在下面的代码示例中，5 和 "Hello World" 都是文本值：

```
// Expression statements.
int i = 5;
string s = "Hello World";
```

有关文本的更多信息，请参见 类型（C# 参考）。

在上面的示例中，i 和 s 都是用于标识局部变量的简单名称。在表达式中使用这些变量时，变量名称计算为当前在该变量的内存位置所存储的值。下面的示例演示了这种情况：

```
int num = 5;
System.Console.WriteLine(num); // Output: 5
num = 6;
System.Console.WriteLine(num); // Output: 6
```

调用表达式

在下面的代码示例中，对 DoWork 的调用是一个调用表达式。

```
DoWork();
```

方法调用要求使用方法的名称（如前面的示例中那样作为名称，或作为其他表达式的结果），后跟括号和任意方法参数。有关更多信息，请参见 方法（C# 编程指南）。委托调用使用委托的名称和括号内的方法参数。有关更多信息，请参见 委托（C# 编程指南）。如果方法返回值，则方法调用和委托调用的计算结果为该方法的返回值。返回 void 的方法不能替代表达式中的值。

查询表达式

有关常规表达式的规则同样适用于查询表达式。有关更多信息，请参见 LINQ 查询表达式（C# 编程指南）。

Lambda 表达式

Lambda 表达式表示没有名称但可以具有输入参数和多个语句的“内联方法”。它们在 LINQ 中广泛用于向方法传递参数。Lambda 表达式被编译为委托或表达式树，具体取决于使用它们的上下文。有关更多信息，请参见 Lambda 表达式（C# 编程

指南）。

表达式树

使用表达式树可以将表达式表示为数据结构。表达式树由 LINQ 提供程序广泛使用，用来将查询表达式转换为在其他某个上下文（如 SQL 数据库）中有意义的代码。有关更多信息，请参见表达式树（C# 和 Visual Basic）。

备注

只要从表达式中识别到变量、对象属性或对象索引器访问，该项的值都会用作表达式的值。在 C# 中，只要表达式的最终计算结果是所需的类型，表达式就可以放置在任何需要值或对象的位置。

重要章节

Variables and Expressions 在 Beginning Visual C# 2010

请参阅

C# 编程指南

方法（C# 编程指南）

委托（C# 编程指南）

运算符（C# 编程指南）

类型（C# 编程指南）

LINQ 查询表达式（C# 编程指南）

运算符（C# 编程指南）

在 C# 中，运算符是应用于表达式或语句中的一个或多个操作数的程序元素。 接受一个操作数的运算符称为一元运算符，例如递增运算符 (**++**) 或 **new**。 接受两个操作数的运算符称为二元运算符，例如算术运算符（**+**、**-**、*****、**/**）。 条件运算符 **?:** 接受三个操作数，是 C# 中唯一的三元运算符。

下面的 C# 语句包含一个一元运算符和一个操作数。 递增运算符 **++** 修改操作数 y 的值。

```
y++;
```

下面的 C# 语句包含两个二元运算符，它们分别有两个操作数。 赋值运算符 **=** 将一个整数变量 y 和一个表达式 2 + 3 作为操作数。 表达式 2 + 3 本身由加法运算符和两个操作数 2 和 3 组成。

```
y = 2 + 3;
```

运算符、计算和运算符优先级

操作数可以是由任何长度的代码组成的有效表达式，且可包含任意数量的子表达式。 在包含多个运算符的表达式中，运算符的应用顺序由运算符优先级、关联性和括号确定。

每个运算符都具有已定义的优先级。 在包含具有不同优先级级别的多个运算符的表达式中，运算符的优先级确定运算符的计算顺序。 例如，下列语句将 3 赋给 n1。

n1 = 11 - 2 * 4;

因为乘法的优先级高于减法，所以首先执行乘法。

下表根据运算符执行的操作类型将它们划分到不同的类别中。 类别按优先级顺序列出。

主要运算符

Expression	描述
x.yx?.y	成员访问条件成员访问
f(x)	方法和委托调用
a[x]a?[x]	数组和索引器访问条件数组和索引器访问
x++	后递增
x--	后递减
new T(...)	对象和委托创建
new T(...){...}	具有初始值设定项的对象创建。 请参阅 对象和集合初始值设定项（C# 编程指南）。
new {...}	匿名对象初始值设定项。 请参阅 匿名类型（C# 编程指南）。
new T[...]	数组创建。 请参阅 数组（C# 编程指南）。
typeof(T)	获取 T 的 System.Type 对象
checked(x)	在已检查的上下文中计算表达式
unchecked(x)	在未检查的上下文中计算表达式
default (T)	获取类型 T 的默认值
delegate {}	匿名函数（匿名方法）

一元运算符

Expression	描述
+x	标识
-x	求反
!x	逻辑求反
~x	按位求反
++x	前递增
--x	前递减
(T)x	将 x 显式转换为类型 T

乘法运算符

Expression	描述
*	乘法
/	除号
%	余数

相加运算符

Expression	描述
x + y	相加、字符串串联、委托组合
x - y	相减、委托移除

移位运算符

Expression	描述
x << y	左移
x >> y	右移

关系和类型运算符

Expression	描述
x < y	小于
x > y	大于
x <= y	小于或等于
x >= y	大于或等于
x is T	如果 x 为 T，则返回 True；否则返回 False。
x as T	返回类型为 T 的 x，如果 x 不是 T，则返回 null

相等运算符

Expression	描述
x == y	等于
x != y	不等于

逻辑、条件和 null 运算符

类别	Expression	描述
逻辑“与”	x & y	整型按位“与”，布尔型逻辑“与”
逻辑 XOR	x ^ y	整型按位 XOR，布尔型逻辑 XOR
逻辑“或”	x | y	整型按位“或”，布尔型逻辑“或”
条件“与”	x && y	仅当 x 为 True 时计算 y
条件“或”	x || y	仅当 x 为 False 时计算 y
null 合并	x ?? y	如果 x 为 Null，则计算结果为 y，否则计算结果为 x
条件运算	x ? y : z	如果 x 为 True，则计算结果为 y；如果 x 为 False 则计算结果为 z

赋值和匿名运算符

Expression	描述
=	赋值
x op= y	复合赋值。 支持以下运算符：+=、-=、*=、/=、%=、&=、|=、!=、<<=, >>=
(T x) => y	匿名函数（lambda 表达式）

结合性

当表达式中出现两个或两个以上具有相同优先级的运算符时，将根据结合性计算它们。 左结合运算符按从左到右的顺序计算。 例如，x *y* / *z* 将计算为 *(x* y) / z。 右结合运算符按从右到左的顺序计算。 例如，赋值运算符是右关联的。 如果不是，下面的代码将导致错误。

```
int a, b, c; c = 1; // The following two lines are equivalent. a =
```

再举一个例子，三元运算符 (?:) 是右结合运算符。 大多数的二元运算符是左结合运算符。

无论表达式中的运算符是左结合运算符还是右结合运算符，都将先从左至右评估各表达式的操作数。 以下示例显示运算符和操作数的计算顺序。

语句	计算顺序
a = b	a、b、=
a = b + c	a、b、c、+、=
a = b + c * d	a、b、c、d、*、+、=
a = b * c + d	a、b、c、*、d、+、=
a = b - c + d	a、b、c、-、d、+、=
a += b -= c	a、b、c、-=、+=

添加括号

可通过使用圆括号更改运算符优先级和相关性。 例如，2 + 3 2 通常计算结果为 *8*，因为乘法运算符的优先级高于加法运算符。 但是，如果你将表达式编写为 *(2 + 3)* 2，则先计算加法，再计算乘法，且结果为 10。 以下示例显示括号表达式中的计算顺序。 如前面的示例中所示，计算操作数之前会应用运算符。

语句	计算顺序
a = (b + c) * d	a、b、c、+、d、*、=
a = b - (c + d)	a、b、c、d、+、-、=
a = (b + c) * (d - e)	a、b、c、+、d、e、-、*、=

运算符重载

对于自定义类和结构，你可以更改运算符的行为。 此过程称为“运算符重载”。 有关更多信息，请参见可重载运算符（C# 编程指南）。

相关章节

有关详细信息，请参阅运算符关键字（C# 参考）和C# 运算符。

请参阅

C# 编程指南

语句、表达式和运算符（C# 编程指南）

匿名函数（C# 编程指南）

匿名函数是一个“内联”语句或表达式，可在需要委托类型的任何地方使用。可以使用匿名函数来初始化命名委托，或传递命名委托（而不是命名委托类型）作为方法参数。

共有两种匿名函数，以下主题中分别讨论了这些函数：

- Lambda 表达式（C# 编程指南）.
- 匿名方法（C# 编程指南）

注意
Lambda 表达式可以绑定到表达式树，也可以绑定到委托。

C# 中委托的发展

在 C# 1.0 中，您通过使用在代码中其他位置定义的方法显式初始化委托来创建委托的实例。C# 2.0 引入了匿名方法的概念，作为一种编写可在委托调用中执行的未命名内联语句块的方式。C# 3.0 引入了 Lambda 表达式，这种表达式与匿名方法的概念类似，但更具表现力并且更简练。这两个功能统称为“匿名函数”。通常，针对 .NET Framework 版本 3.5 及更高版本的应用程序应使用 Lambda 表达式。

下面的示例演示了从 C# 1.0 到 C# 3.0 委托创建过程的发展：

```
class Test
{
    delegate void TestDelegate(string s);
    static void M(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        // Original delegate syntax required
        // initialization with a named method.
        TestDelegate testDelA = new TestDelegate(M);

        // C# 2.0: A delegate can be initialized with
        // inline code, called an "anonymous method." This
        // method takes a string as an input parameter.
        TestDelegate testDelB = delegate(string s) { Console.WriteI

        // C# 3.0\. A delegate can be initialized with
        // a lambda expression. The lambda also takes a string
        // as an input parameter (x). The type of x is inferred by
        TestDelegate testDelC = (x) => { Console.WriteLine(x); };

        // Invoke the delegates.
        testDelA("Hello. My name is M and I write lines.");
        testDelB("That's nothing. I'm anonymous and ");
        testDelC("I'm a famous author.");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Hello. My name is M and I write lines.
    That's nothing. I'm anonymous and
    I'm a famous author.
    Press any key to exit.
 */
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

语句、表达式和运算符（C# 编程指南）

Lambda 表达式（C# 编程指南）

委托（C# 编程指南）

表达式树（C# 和 Visual Basic）

可重载运算符（C# 编程指南）

C# 通过使用 operator 关键字定义静态成员函数，来允许用户定义的类型重载运算符。不过并非所有运算符都可以进行重载，并且其他运算符具有限制，如下表所列：

<table>
<tr><th>运算符</th></tr>
<tr><td>+、-、!、~、++、--、true、false</td></tr>
<tr><td>+, -, *, /, %, &,|, ^, <<, >></td></tr>
<tr><td>==, !=, <, >, <=, >=</td></tr>
<tr><td>&&, ||</td></tr>
<tr><td>[]</td></tr>
<tr><td>(T)x</td></tr>
</table>

+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
=、.、?:、??、->、=>、f(x)、as、checked、unchecked、default、delegate、is、new、sizeof、

> 注意
>
> 如果进行重载，则比较运算符必须成对进行重载；也就是说，如果 **==** 进行重载，则 **!=** 也必须进行重载。反之亦然，对于 **<** 和 **>** 以及 **<=** 和 **>=** 也是类似情况。

若要在自定义类上重载运算符，需要在该类上创建具有正确签名的方法。该方法必须命名为“运算符 X”，其中 X 是重载的运算符的名称或符号。一元运算符具有一个参数，二元运算符具有两个参数。在每种情况下，都必须有一个参数与声明运算符的类或结构的类型相同。

```
public static Complex operator +(Complex c1, Complex c2)
    {
        Return new Complex(c1.real + c2.real, c1.imaginary + c2.ima
    }
```

直接只返回表达式结果的定义很常见。对于这些情况，有一种使用 **=>** 的语法快捷方式。

```
public static Complex operator +(Complex c1, Complex c2) =>
        new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary)

    // Override ToString() to display a complex number
    // in the traditional format:
    public override string ToString() => $"{this.real} + {this.imag
```

有关详细新，请参阅如何：使用运算符重载创建复数类（C# 编程指南）。

请参阅

C# 编程指南

语句、表达式和运算符（C# 编程指南）

运算符（C# 编程指南）

C# 运算符

重载的运算符为何在 C# 中始终为静态？

转换运算符（C# 编程指南）

C# 允许程序员在类或结构上声明转换，以便类或结构与其他类或结构或者基本类型进行相互转换。转换的定义方法类似于运算符，并根据它们所转换到的类型命名。要转换的参数类型或转换结果的类型必须是（不能两者同时都是）包含类型。

```
class SampleClass
{
    public static explicit operator SampleClass(int i)
    {
        SampleClass temp = new SampleClass();
        // code to convert from int to SampleClass...

        return temp;
    }
}
```

转换运算符概述

转换运算符具有以下特点：

- 声明为 **implicit** 的转换在需要时自动进行。
- 声明为 **explicit** 的转换需要调用强制转换。
- 所有转换都必须声明为 **static**。

相关章节

有关更多信息：

- 使用转换运算符（C# 编程指南）
- 强制转换和类型转换（C# 编程指南）
- 如何：在结构间实现用户定义的转换（C# 编程指南）
- explicit（C# 参考）
- implicit（C# 参考）
- static（C# 参考）

请参阅

Convert

C# 编程指南

在 C# 中链接的用户定义的显式转换

如何：使用运算符重载创建复数类（C# 编程指南）

本示例展示如何使用运算符重载创建定义复数加法的复数类 Complex。本程序使用 **ToString** 方法的重载显示数字的虚部和实部以及加法结果。

```
public struct Complex
{
    public int real;
    public int imaginary;

    // Constructor.
    public Complex(int real, int imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    // Specify which operator to overload (+),
    // the types that can be added (two Complex objects),
    // and the return type (Complex).
    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex(c1.real + c2.real, c1.imaginary + c2.ima
    }

    // Override the ToString() method to display a complex number
    // in the traditional format:
    public override string ToString()
    {
        return (System.String.Format("{0} + {1}i", real, imaginary)
    }
}

class TestComplex
{
    static void Main()
    {
        Complex num1 = new Complex(2, 3);
        Complex num2 = new Complex(3, 4);

        // Add two Complex objects by using the overloaded + operat
        Complex sum = num1 + num2;

        // Print the numbers and the sum by using the overridden
        // ToString method.
        System.Console.WriteLine("First complex number:  {0}", num1
        System.Console.WriteLine("Second complex number: {0}", num2
        System.Console.WriteLine("The sum of the two numbers: {0}",
```

```
            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
    /* Output:
        First complex number:  2 + 3i
        Second complex number: 3 + 4i
        The sum of the two numbers: 5 + 7i
    */
```

请参阅

C# 编程指南

C# 运算符

运算符（C# 参考）

Why are overloaded operators always static in C#?（在 C# 中，为什么重载的运算符始终是静态的？）

相等比较（C# 编程指南）

有时必须比较两个值是否相等。在某些情况下，您测试的是“值相等性”（也称为“等效性”），意即两个变量包含的值相等。而在其他情况下，则必须确定两个变量是否引用内存中的同一基础对象。这种类型的相等性称为“引用相等性”（或“标识”）。本主题描述这两种相等性，并提供指向其他主题的链接以了解更多信息。

引用相等性

引用相等性是指两个对象引用均引用同一基础对象。这可以通过简单的赋值来实现，如下面的示例所示。

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", are

        // Assign b to a.
        b = a;

        // Repeat calls with different results.
        areEqual = System.Object.ReferenceEquals(a, b);
        // True:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", are

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

在此代码中，创建了两个对象，但在赋值语句后，这两个引用所引用的是同一对象。因此，它们具有引用相等性。使用 ReferenceEquals 方法来确定两个引用是否引用同一对象。

引用相等性的概念仅适用于引用类型。由于在将值类型的实例赋给变量时将建立值的副本，因此值类型对象无法具有引用相等性。因此，永远不能有引用内存中的同一位置的两个未装箱结构。此外，如果您使用 ReferenceEquals 来比较两个值类型，结果将始终为 **false**，即使对象中包含的值都相同也是如此。这是因为每个变量都会被装箱到单独的对象实例中。有关更多信息，请参见如何：测试引用相等性（标识）（C# 编程指南）。

值相等性

值相等性是指两个对象包含相同的一个或多个值。对于基元值类型（例如 int 或 bool），针对值相等性的测试简单明了。您可以使用 == 运算符，如下面的示例所示。

```
int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if( b == a)
{
    // The two integers are equal.
}
```

对于大多数其他类型，针对值相等性的测试较为复杂，因为它需要您了解类型对值相等性的定义方式。对于具有多个字段或属性的类和结构，值相等性的定义通常是指所有字段或属性都具有相同的值。例如，如果 pointA.X 等于 pointB.X，并且 pointA.Y 等于 pointB.Y，则可以将两个 **Point** 对象定义为相等。

不过，并不要求类型中的所有字段均相等。只需子集相等即可。在比较不属于您的类型时，应确保明确了解相等性对于该类型是如何定义的。有关如何在您自己的类和结构中定义值相等性的更多信息，请参见如何：为类型定义值相等性（C# 编程指南）。

浮点值的值相等性

由于二进制计算机上浮点算法的不精确性，因此浮点值（double 和 float）的相等比较会出现问题。有关更多信息，请参见主题 System.Double 中的备注。

相关主题

标题	描述
如何：测试引用相等性（标识）（C# 编程指南）	描述如何确定两个变量是否具有引用相等性。
如何：为类型定义值相等性（C# 编程指南）	描述如何为类型提供值相等性的自定义定义。
C# 编程指南	提供一些链接，这些链接指向有关重要的 C# 语言功能以及通过 .NET Framework 提供给 C# 的功能的详细信息。
类型（C# 编程指南）	提供有关 C# 类型系统的信息以及指向附加信息的链接。

请参阅

C# 编程指南

字符串（C# 编程指南）

字符串是 String 类型的对象，它的值是文本。在内部，文本被存储为 Char 对象的顺序只读集合。C# 字符串末尾没有以 null 结尾的字符；因此 C# 字符串可以包含任意数目的嵌入式 null 字符（“\0”）。字符串的 Length 属性代表它包含的 **Char** 对象的数量，而不是 Unicode 字符的数量。若要访问字符串中的各个 Unicode 码位，请使用 StringInfo 对象。

字符串与System.String

在 C# 中，**string** 关键字是 String 的别名。因此，**String** 与 **string** 等效，您可以根据自己的喜好选择命名约定。 **String** 类提供了很多用于安全地创建、操作和比较字符串的方法。此外，C# 语言还重载某些运算符来简化常见的字符串操作。有关关键字的更多信息，请参见string（C# 参考）。有关类型及其方法的更多信息，请参见 String。

声明和初始化字符串

可以通过各种方式来声明和初始化字符串，如下面的示例所示：

```
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

//Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

注意，除了在使用字符数组初始化字符串时以外，不要使用 new 运算符创建字符串对象。

使用 Empty 常量值初始化字符串可新建字符串长度为零的 String 对象。零长度字符串的字符串表示形式为 ""。使用 Empty 值（而不是 null）初始化字符串可以降低发生 NullReferenceException 的可能性。请在尝试访问字符串之前使用静态 IsNullOrEmpty(String) 方法验证字符串的值。

字符串对象的不可变性

字符串对象是不可变的：即它们创建之后就无法更改。所有看似修改字符串的 String 方法和 C# 运算符实际上都以新字符串对象的形式返回结果。在下面的示例中，当连接 s1 和 s2 的内容以形成一个字符串时，不会修改两个原始字符串。 **+=** 运算符会创建一个包含组合内容的新字符串。这个新对象赋给变量 s1，而最初赋给 s1 的对象由于没有其他任何变量包含对它的引用而释放，用于垃圾回收。

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2\. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

由于“修改”字符串实际上是创建新字符串，因此创建对字符串的引用时必须谨慎。如果创建了对字符串的引用，然后“修改”原始字符串，则该引用指向的仍是原始对象，而不是修改字符串时创建的新对象。下面的代码说明了这种行为：

```
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
//Output: Hello
```

有关如何创建基于修改（例如搜索和替换原始字符串的操作）的新字符串的更多信息，请参见如何：修改字符串内容（C# 编程指南）。

正则字符串和原义字符串

如果必须嵌入 C# 提供的转义符，则应使用正则字符串，如下面的示例所示：

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
  Row 1
  Row 2
  Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Colerid
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

如果字符串文本包含反斜杠字符（例如在文件路径中），为方便起见和提高可读性，应使用原义字符串。由于原义字符串保留换行符作为字符串文本的一部分，因此可用于初始化多行字符串。在原义字符串中嵌入引号时请使用双引号。下面的示

例演示原义字符串的一些常见用途：

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
   Thus on mine arm, most soothing sweet it is
   To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."
```

字符串转义序列

转义序列	字符名称	**Unicode 编码**
\'	单引号	0x0027
\"	双引号	0x0022
\	反斜杠	0x005C
\0	Null	0x0000
\a	警报	0x0007
\b	Backspace	0x0008
\f	换页	0x000C
\n	换行	0x000A
\r	回车	0x000D
\t	水平制表符	0x0009
\U	代理项对的 Unicode 转义序列。	\Unnnnnnnn
\u	Unicode 转义序列	\u0041 = "A"
\v	垂直制表符	0x000B
\x	Unicode 转义序列类似于“\u”，只是长度可变。	\x0041 = "A"

> 注意
>
> 编译时，原义字符串转换为所有转义序列均保持不变的普通字符串。因而，如果在调试器监视窗口中查看原义字符串，则看到的将是编译器添加的转义字符，而不是源代码中的原义版本。例如，原义字符串 @"C:\files.txt" 在监视窗口中将显示为 "C:\\files.txt"。

格式字符串

格式字符串是内容可以在运行时动态确定的一种字符串。采用以下方式创建格式字符串：使用静态 Format 方法并在大括号中嵌入占位符，这些占位符将在运行时替换为其他值。下面的示例使用格式字符串输出循环中每个迭代的结果：

```
class FormatString
{
    static void Main()
    {
        // Get user input.
        System.Console.WriteLine("Enter a number");
        string input = System.Console.ReadLine();

        // Convert the input string to an int.
        int j;
        System.Int32.TryParse(input, out j);

        // Write a different string each iteration.
        string s;
        for (int i = 0; i < 10; i++)
        {
            // A simple format string with no alignment formatting
            s = System.String.Format("{0} times {1} = {2}", i, j, (
            System.Console.WriteLine(s);
        }

        //Keep the console window open in debug mode.
        System.Console.ReadKey();
    }
}
```

WriteLine 方法的一个重载将格式字符串用作参数。因此，可以只嵌入格式字符串，而无需显式调用该方法。但若使用 WriteLine 方法在 Visual Studio“输出”窗口中显示调试输出，则必须显式调用 Format 方法，因为 WriteLine 只接受字符串，而不接受格式字符串。有关格式字符串的更多信息，请参见.NET Framework 中的格式化类型。

子字符串

子字符串是包含在字符串中的任意字符序列。使用 Substring 方法可以基于原始字符串的一部分创建新字符串。可以使用 IndexOf 方法搜索子字符串的一个或多个匹配项。使用 Replace 方法可将指定子字符串的所有匹配项替换为一个新字符串。与 Substring 方法一样，Replace 实际上返回的也是新字符串，而不修改原始字符串。有关更多信息，请参见如何：使用字符串方法搜索字符串（C# 编程指南）和如何：修改字符串内容（C# 编程指南）。

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

访问各个字符

可以使用带索引值的数组表示法获取对各个字符的只读访问，如下面的示例所示：

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

如果 String 方法不提供修改字符串中的各个字符所必须具有的功能，则您可以使用 StringBuilder 对象“就地”修改各个字符，然后使用 StringBuilder 方法创建一个新字符串来存储结果。在下面的示例中，假设您必须以特定方式修改原始字符串，然后存储结果以备将来使用：

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS lOCK
System.Text.StringBuilder sb = new System.Text.StringBuilder(questi

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Null 字符串和空字符串

空字符串是不包含字符的 System.String 对象的实例。在各种编程方案中经常会使用空字符串表示空白文本字段。可以对空字符串调用方法，因为它们是有效的 System.String 对象。空字符串可按如下方式初始化：

```
string s = String.Empty;
```

相反，null 字符串并不引用 System.String 对象的实例，任何对 null 字符串调用方法的尝试都会生成 NullReferenceException。但是，可以在串联和比较操作中将 null 字符串与其他字符串一起使用。下面的示例阐释了引用 null 字符串导致引发异常的情形以及并不导致引发异常的情形：

```
static void Main()
{
    string str = "hello";
    string nullStr = null;
    string emptyStr = String.Empty;

    string tempStr = str + nullStr;
    // Output of the following line: hello
    Console.WriteLine(tempStr);

    bool b = (emptyStr == nullStr);
    // Output of the following line: False
    Console.WriteLine(b);

    // The following line creates a new empty string.
    string newStr = emptyStr + nullStr;

    // Null strings and empty strings behave differently. The follo
    // two lines display 0.
    Console.WriteLine(emptyStr.Length);
    Console.WriteLine(newStr.Length);
    // The following line raises a NullReferenceException.
    //Console.WriteLine(nullStr.Length);

    // The null character can be displayed and counted, like other
    string s1 = "\x0" + "abc";
    string s2 = "abc" + "\x0";
    // Output of the following line: * abc*
    Console.WriteLine("*" + s1 + "*");
    // Output of the following line: *abc *
    Console.WriteLine("*" + s2 + "*");
    // Output of the following line: 4
    Console.WriteLine(s2.Length);
}
```

使用 **StringBuilder** 快速创建字符串

.NET 中的字符串操作已高度优化，大多数情况下不会显著影响性能。但在某些应用场景中，例如在执行好几百甚至好几千次的紧凑循环中，字符串操作会影响性能。 StringBuilder 类创建了一个字符串缓冲区，用于在程序执行大量字符串操作时提供更好的性能。 StringBuilder 字符串还使您能够重新分配个别字符（内置字符串数据类型所不支持的字符）。例如，此代码在不创建新字符串的情况下更改了一个字符串的内容：

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat:
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//Outputs Cat: the ideal pet
```

在本示例中，StringBuilder 对象用于从一组数值类型中创建字符串：

```
class TestStringBuilder
{
    static void Main()
    {
        System.Text.StringBuilder sb = new System.Text.StringBuilde

        // Create a string composed of numbers 0 - 9
        for (int i = 0; i < 10; i++)
        {
            sb.Append(i.ToString());
        }
        System.Console.WriteLine(sb);  // displays 0123456789

        // Copy one character of the string (not possible with a Sy
        sb[0] = sb[9];

        System.Console.WriteLine(sb);  // displays 9123456789
    }
}
```

字符串、扩展方法和 LINQ

由于 String 类型实现 IEnumerable<T>，因此可以对字符串使用 Enumerable 类中定义的扩展方法。对于 String 类型，为了避免视觉上的混乱，从 IntelliSense 中排除了这些方法，但这些方法仍然可用。此外，还可以对字符串使用 LINQ 查询表达式。有关更多信息，请参见 LINQ and Strings。

相关主题

主题	说明
如何：修改字符串内容（C# 编程指南）	提供一个代码示例来演示如何修改字符串的内容。
如何：串联多个字符串（C# 编程指南）	阐释如何使用 **+** 运算符和 **Stringbuilder** 类在编译时和运行时将字符串联接在一起。
如何：比较字符串（C# 编程指南）	演示如何执行字符串的序号比较。
如何：拆分字符串（C# 编程指南）	包含一个代码示例，该示例演示如何使用 **String.Split** 方法来分析字符串。
如何：使用字符串方法搜索字符串（C# 编程指南）	解释如何使用特定方法来搜索字符串。
如何：使用正则表达式搜索字符串（C# 编程指南）	解释如何使用正则表达式来搜索字符串。
如何：确定字符串是否表示数值（C# 编程指南）	演示如何安全地分析字符串以了解其是否具有有效数值。
如何：将字符串转换为 DateTime（C# 编程指南）	演示如何将诸如“01/24/2008”这样的字符串转换为 System.DateTime 对象。
.NET Framework 中的基本字符串操作	提供一些指向主题的链接，这些主题使用 System.String 和 System.Text.StringBuilder 方法来执行基本字符串操作。
在 .NET Framework 中分析字符串	介绍如何将字符或空格插入到字符串中。
在 .NET Framework 中比较字符串	包含有关如何比较字符串的信息，并且提供使用 C# 和 Visual Basic 编写的示例。
在 .NET Framework 中使用 StringBuilder 类	描述如何使用 StringBuilder 类创建和修改动态字符串对象。
LINQ and Strings	提供有关如何使用 LINQ 查询来执行各种字符串操作的信息。
C# 编程指南	提供指向解释 C# 中编程构造的主题的链接。

重要章节

More About Variables 在 Beginning Visual C# 2010

如何：串联多个字符串（C# 编程指南）

串联是将一个字符串追加到另一个字符串末尾的过程。使用 **+** 运算符串联字符串文本或字符串常量时，编译器会创建一个字符串。串联不在运行时发生。但字符串变量只能在运行时串联，对此，您应该了解各种方法的性能含义。

下面的示例演示如何将一个长字符串拆分为几个较短的字符串，从而提高源代码的可读性。这些较短的字符串将在编译时串联成一个字符串。无论涉及到多少个字符串，都不会有运行时性能开销。

```
static void Main()
{
    // Concatenation of literals is performed at compile time, not
    string text = "Historically, the world of data and the world of
    "have not been well integrated. Programmers work in C# or Visua
    "and also in SQL or XQuery. On the one side are concepts such a
    "objects, fields, inheritance, and .NET Framework APIs. On the
    "are tables, columns, rows, nodes, and separate languages for d
    "them. Data types often require translation between the two wor
    "different standard functions. Because the object world has no
    "query can only be represented as a string without compile-time
    "IntelliSense support in the IDE. Transferring data from SQL ta
    "objects in memory is often tedious and error-prone.";

    Console.WriteLine(text);
}
```

若要串联字符串变量，可以使用 **+** 或 **+=** 运算符，也可以使用 String.Concat、String.Format 或 StringBuilder.Append 方法。 **+** 运算符容易使用，且有利于提高代码的直观性。即使在一条语句中使用多个 + 运算符，字符串内容也将只复制一次。但是，如果重复此操作多次（如使用循环），则可能会导致出现效率问题。例如，考虑下面的代码：

```
static void Main(string[] args)
{
    // To run this program, provide a command line string.
    // In Visual Studio, see Project > Properties > Debug.
    string userName = args[0];
    string date = DateTime.Today.ToShortDateString();

    // Use the + and += operators for one-time concatenations.
    string str = "Hello " + userName + ". Today is " + date + ".";
    System.Console.WriteLine(str);

    str += " How are you today?";
    System.Console.WriteLine(str);

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

// Example output:
//  Hello Alexander. Today is 1/22/2008.
//  Hello Alexander. Today is 1/22/2008\. How are you today?
//  Press any key to exit.
//
```

注意

在字符串串联操作中，C# 编译器对 null 字符串和空字符串进行相同的处理，但它不转换原始 null 字符串的值。

如果您串联的字符串数量不那么巨大（例如，在循环中），那么这些代码的性能成本可能不会很高。上述情况同样适用于 String.Concat 和 String.Format 方法。

但如果性能的优劣很重要，则应该总是使用 StringBuilder 类来串联字符串。下面的代码使用 StringBuilder 类的 Append 方法来串联字符串，因此不会有 + 运算符的链接作用产生。

```
class StringBuilderTest
{
    static void Main()
    {
        string text = null;

        // Use StringBuilder for concatenation in tight loops.
        System.Text.StringBuilder sb = new System.Text.StringBuilde
        for (int i = 0; i < 100; i++)
        {
            sb.AppendLine(i.ToString());
        }
        System.Console.WriteLine(sb.ToString());

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
// Output:
// 0
// 1
// 2
// 3
// 4
// ...
//
```

请参阅

String

StringBuilder

C# 编程指南

字符串（C# 编程指南）

如何：修改字符串内容（C# 编程指南）

由于字符串是不可变的，因此一个字符串对象一旦创建，值就不能再更改（在不使用不安全代码的情况下）。不过，修改字符串的值然后将结果存储到新的字符串对象中有很多种方法。 System.String 类提供作用于输入字符串并返回新字符串对象的方法。在很多情况下，可以将这个新对象赋给保存原始字符串的变量。 System.Text.RegularExpressions.Regex 类提供其他一些以类似方式工作的方法。 System.Text.StringBuilder 类提供一个可“就地”修改的字符缓冲区。调用 StringBuilder.ToString 方法可新建包含此缓冲区的当前内容的字符串对象。

下面的示例演示替换或移除指定字符串中的子字符串的各种方法。

```
class ReplaceSubstrings
{
    string searchFor;
    string replaceWith;

    static void Main(string[] args)
    {

        ReplaceSubstrings app = new ReplaceSubstrings();
        string s = "The mountains are behind the clouds today.";

        // Replace one substring with another with String.Replace.
        // Only exact matches are supported.
        s = s.Replace("mountains", "peaks");
        Console.WriteLine(s);
        // Output: The peaks are behind the clouds today.

        // Use Regex.Replace for more flexibility.
        // Replace "the" or "The" with "many" or "Many".
        // using System.Text.RegularExpressions
        app.searchFor = "the"; // A very simple regular expression
        app.replaceWith = "many";
        s = Regex.Replace(s, app.searchFor, app.ReplaceMatchCase, F
        Console.WriteLine(s);
        // Output: Many peaks are behind many clouds today.

        // Replace all occurrences of one char with another.
        s = s.Replace(' ', '_');
        Console.WriteLine(s);
        // Output: Many_peaks_are_behind_many_clouds_today.

        // Remove a substring from the middle of the string.
        string temp = "many_";
        int i = s.IndexOf(temp);
        if (i >= 0)
        {
            s = s.Remove(i, temp.Length);
```

```
        }
        Console.WriteLine(s);
        // Output: Many_peaks_are_behind_clouds_today.

        // Remove trailing and leading whitespace.
        // See also the TrimStart and TrimEnd methods.
        string s2 = "    I'm wider than I need to be.      ";
        // Store the results in a new string variable.
        temp = s2.Trim();
        Console.WriteLine(temp);
        // Output: I'm wider than I need to be.

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Custom match method called by Regex.Replace
    // using System.Text.RegularExpressions
    string ReplaceMatchCase(Match m)
    {
        // Test whether the match is capitalized
        if (Char.IsUpper(m.Value[0]) == true)
        {
            // Capitalize the replacement string
            // using System.Text;
            StringBuilder sb = new StringBuilder(replaceWith);
            sb[0] = (Char.ToUpper(sb[0]));
            return sb.ToString();
        }
        else
        {
            return replaceWith;
        }
    }
}
```

若要使用数组表示法访问字符串中的各个字符，可以使用 StringBuilder 对象，该对象重载 [] 运算符以提供对其内部字符缓冲区的访问。也可以使用 ToCharArray 方法将该字符串转换为一个字符数组。下面的示例使用 **ToCharArray** 创建该数组。然后修改该数组中的某些元素。之后再调用采用一个字符数组作为输入参数的字符串构造函数来创建一个新字符串。

```
class ModifyStrings
{
    static void Main()
    {
        string str = "The quick brown fox jumped over the fence";
        System.Console.WriteLine(str);

        char[] chars = str.ToCharArray();
        int animalIndex = str.IndexOf("fox");
        if (animalIndex != -1)
        {
            chars[animalIndex++] = 'c';
            chars[animalIndex++] = 'a';
            chars[animalIndex] = 't';
        }

        string str2 = new string(chars);
        System.Console.WriteLine(str2);

        // Keep the console window open in debug mode
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
  The quick brown fox jumped over the fence
  The quick brown cat jumped over the fence
*/
```

下面的示例针对的是一种非常罕见的情况，即您可能希望使用不安全代码以类似于 C 样式字符数组的方式就地修改字符串。此示例演示如何使用 fixed 关键字“就地”访问各个字符。此外还演示对字符串进行不安全操作可能产生的一个副作用，此副作用是由于 C# 编译器在内部存储（暂存）字符串的方式而导致的。通常，除非绝对必要，否则不应该使用这种方法。

```
class UnsafeString
{
    unsafe static void Main(string[] args)
    {
        // Compiler will store (intern)
        // these strings in same location.
        string s1 = "Hello";
        string s2 = "Hello";

        // Change one string using unsafe code.
        fixed (char* p = s1)
        {
            p[0] = 'C';
        }

        //  Both strings have changed.
        Console.WriteLine(s1);
        Console.WriteLine(s2);

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

}
```

请参阅

C# 编程指南

字符串（C# 编程指南）

如何：比较字符串（C# 编程指南）

比较字符串时，产生的结果会是一个字符串大于或小于另一个字符串，或者两个字符串相等。根据执行的是序号比较还是区分区域性比较，确定结果时所依据的规则会有所不同。对特定的任务使用正确类型的比较十分重要。

当需要对两个字符串的值进行比较和排序而不需要考虑语言惯例时，请使用基本的序号比较。基本的序号比较 (**System.StringComparison.Ordinal**) 是区分大小写的，这意味着两个字符串的字符必须完全匹配："and"不等于"And"或"AND"。常用的变量有 **System.StringComparison.OrdinalIgnoreCase**，它将匹配"and"、"And"和"AND"；还有 **StringComparison.OrdinalIgnoreCase**，它常用于比较文件名、路径名和网络路径，以及其值不随用户计算机的区域设置的更改而变化的任何其他字符串。有关更多信息，请参见System.StringComparison。

区分区域性比较通常用于对终端用户输入的字符串进行比较和排序，因为这些字符串的字符和排序约定可能会根据用户计算机的区域设置的不同而有所不同。即使是包含相同字符的字符串，也可能会根据当前线程的区域性不同而不同。

> 注意
>
> 在比较字符串时，您使用的方法应该显式指定了要执行的比较类型。这可增强代码的可维护性和可读性。应尽可能使用具有 StringComparison 枚举参数的 System.String 和 System.Array 类的方法重载，以便可以指定要执行的比较类型。比较字符串时，最好避免使用 **==** 和 **!=** 运算符。还应该避免使用 String.CompareTo 实例方法，因为这些重载没有 StringComparison。

下面的示例演示如何正确地比较其值将不随用户计算机的区域设置的改变而变化的字符串。此外，它还演示 C# 中的字符串限定功能。如果程序声明了两个或更多个相同的字符串变量，编译器会将它们存储在同一位置。通过调用 ReferenceEquals 方法，可以看到这两个字符串实际引用内存中的同一对象。使用 String.Copy 方法可避免此限定，如下面的示例所示。

```
// Internal strings that will never be localized.
string root = @"C:\users";
string root2 = @"C:\Users";

// Use the overload of the Equals method that specifies a StringCor
// Ordinal is the fastest way to compare two strings.
bool result = root.Equals(root2, StringComparison.Ordinal);

Console.WriteLine("Ordinal comparison: {0} and {1} are {2}", root,
                    result ? "equal." : "not equal.");

// To ignore case means "user" equals "User". This is the same as u
// String.ToUpperInvariant on each string and then performing an or
result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
Console.WriteLine("Ordinal ignore case: {0} and {1} are {2}", root,
                     result ? "equal." : "not equal.");

// A static method is also available.
bool areEqual = String.Equals(root, root2, StringComparison.Ordinal

// String interning. Are these really two distinct objects?
string a = "The computer ate my source code.";
string b = "The computer ate my source code.";

// ReferenceEquals returns true if both objects
// point to the same location in memory.
if (String.ReferenceEquals(a, b))
    Console.WriteLine("a and b are interned.");
else
    Console.WriteLine("a and b are not interned.");

// Use String.Copy method to avoid interning.
string c = String.Copy(a);

if (String.ReferenceEquals(a, c))
    Console.WriteLine("a and c are interned.");
else
    Console.WriteLine("a and c are not interned.");

// Output:
// Ordinal comparison: C:\users and C:\Users are not equal.
// Ordinal ignore case: C:\users and C:\Users are equal.
// a and b are interned.
// a and c are not interned.
```

下面的示例演示如何通过首选方式比较字符串，该首选方式使用具有 StringComparison 枚举的 System.String 方法。请注意，这里没有使用 String.CompareTo 实例方法，因为这些重载没有 StringComparison。

```
// "They dance in the street."
// Linguistically (in Windows), "ss" is equal to
// the German essetz: 'ß' character in both en-US and de-DE culture
string first = "Sie tanzen in die Straße.";
string second = "Sie tanzen in die Strasse.";

Console.WriteLine("First sentence is {0}", first);
Console.WriteLine("Second sentence is {0}", second);

// Store CultureInfo for the current culture. Note that the origina
// can be set and retrieved on the current thread object.
System.Threading.Thread thread = System.Threading.Thread.CurrentThr
System.Globalization.CultureInfo originalCulture = thread.CurrentCu

// Set the culture to en-US.
thread.CurrentCulture = new System.Globalization.CultureInfo("en-US

// For culture-sensitive comparisons, use the String.Compare
// overload that takes a StringComparison value.
int i = String.Compare(first, second, StringComparison.CurrentCultu
Console.WriteLine("Comparing in {0} returns {1}.", originalCulture.

// Change the current culture to Deutch-Deutchland.
thread.CurrentCulture = new System.Globalization.CultureInfo("de-DE
i = String.Compare(first, second, StringComparison.CurrentCulture);
Console.WriteLine("Comparing in {0} returns {1}.", thread.CurrentCu

// For culture-sensitive string equality, use either StringCompare
// or the String.Equals overload that takes a StringComparison valu
thread.CurrentCulture = originalCulture;
bool b = String.Equals(first, second, StringComparison.CurrentCultu
Console.WriteLine("The two strings {0} equal.", b == true ? "are"

/*
 * Output:
    First sentence is Sie tanzen in die Straße.
    Second sentence is Sie tanzen in die Strasse.
    Comparing in en-US returns 0.
    Comparing in de-DE returns 0.
    The two strings are equal.
 */
```

下面的示例演示如何使用具有 System.StringComparer 参数的静态 Array 方法以区分区域性的方式对数组中的字符串进行排序和搜索。

```
class SortStringArrays
{
    static void Main()
    {
```

```
        string[] lines = new string[]
        {
            @"c:\public\textfile.txt",
            @"c:\public\textFile.TXT",
            @"c:\public\Text.txt",
            @"c:\public\testfile2.txt"
        };

        Console.WriteLine("Non-sorted order:");
        foreach (string s in lines)
        {
            Console.WriteLine("   {0}", s);
        }

        Console.WriteLine("\n\rSorted order:");

        // Specify Ordinal to demonstrate the different behavior.
        Array.Sort(lines, StringComparer.Ordinal);

        foreach (string s in lines)
        {
            Console.WriteLine("   {0}", s);
        }

        string searchString = @"c:\public\TEXTFILE.TXT";
        Console.WriteLine("Binary search for {0}", searchString);
        int result = Array.BinarySearch(lines, searchString, Strin
        ShowWhere<string>(lines, result);

        //Console.WriteLine("{0} {1}", result > 0 ? "Found" : "Did

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    // Displays where the string was found, or, if not found,
    // where it would have been located.
    private static void ShowWhere<T>(T[] array, int index)
    {
        if (index < 0)
        {
            // If the index is negative, it represents the bitwise
            // complement of the next larger element in the array.
            index = ~index;

            Console.Write("Not found. Sorts between: ");

            if (index == 0)
                Console.Write("beginning of array and ");
            else
                Console.Write("{0} and ", array[index - 1]);
```

```
            if (index == array.Length)
                Console.WriteLine("end of array.");
            else
                Console.WriteLine("{0}.", array[index]);
        }
        else
        {
            Console.WriteLine("Found at index {0}.", index);
        }
    }

}
/*
 * Output:
    Non-sorted order:
       c:\public\textfile.txt
       c:\public\textFile.TXT
       c:\public\Text.txt
       c:\public\testfile2.txt

    Sorted order:
       c:\public\Text.txt
       c:\public\testfile2.txt
       c:\public\textFile.TXT
       c:\public\textfile.txt
    Binary search for c:\public\TEXTFILE.TXT
    Found at index 2.
 */
```

当元素或键的类型为 **string** 时，System.Collections.Hashtable、System.Collections.Generic.Dictionary<TKey, TValue> 和 System.Collections.Generic.List<T> 等集合类的构造函数具有 System.StringComparer 参数。通常，应尽可能使用这些构造函数，并指定 Ordinal 或 OrdinalIgnoreCase。

请参阅

System.Globalization.CultureInfo

System.StringComparer

字符串（C# 编程指南）

在 .NET Framework 中比较字符串

对应用程序进行全球化和本地化

如何：拆分字符串（C# 编程指南）

下面的代码示例演示如何使用 String.Split 方法分析字符串。作为输入，Split 采用一个字符数组指示哪些字符被用作分隔符。本示例中使用了空格、逗号、句点、冒号和制表符。一个含有这些分隔符的数组被传递给 Split，并使用结果字符串数组分别显示句子中的每个单词。

```
class TestStringSplit
{
    static void Main()
    {
        char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

        string text = "one\ttwo three:four,five six seven";
        System.Console.WriteLine("Original text: '{0}'", text);

        string[] words = text.Split(delimiterChars);
        System.Console.WriteLine("{0} words in text:", words.Lengtl

        foreach (string s in words)
        {
            System.Console.WriteLine(s);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Original text: 'one     two three:four,five six seven'
    7 words in text:
    one
    two
    three
    four
    five
    six
    seven
 */
```

请参阅

C# 编程指南

字符串（C# 编程指南）

.NET Framework 正则表达式

如何：使用字符串方法搜索字符串（C# 编程指南）

string 类型（它是 System.String 类的别名）为搜索字符串的内容提供了许多有用的方法。

下面的示例使用 IndexOf、LastIndexOf、StartsWith 和 EndsWith 方法搜索字符串。

```
class StringSearch
{
    static void Main()
    {
        string str = "Extension methods have all the capabilities c

        // Write the string and include the quotation marks.
        System.Console.WriteLine("\"{0}\"", str);

        // Simple comparisons are always case sensitive!
        bool test1 = str.StartsWith("extension");
        System.Console.WriteLine("Starts with \"extension\"? {0}",

        // For user input and strings that will be displayed to the
        // use the StringComparison parameter on methods that have
        bool test2 = str.StartsWith("extension", System.StringCompa
        System.Console.WriteLine("Starts with \"extension\"? {0} (i

        bool test3 = str.EndsWith(".", System.StringComparison.Curr
        System.Console.WriteLine("Ends with '.'? {0}", test3);

        // This search returns the substring between two strings, s
        // the first index is moved to the character just after the
        int first = str.IndexOf("methods") + "methods".Length;
        int last = str.LastIndexOf("methods");
        string str2 = str.Substring(first, last - first);
        System.Console.WriteLine("Substring between \"methods\" and

        // Keep the console window open in debug mode
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/*
Output:
"Extension methods have all the capabilities of regular static meth
Starts with "extension"? False
Starts with "extension"? True (ignoring case)
Ends with '.'? True
Substring between "methods" and "methods": ' have all the capabilit
Press any key to exit.
*/
```

请参阅

C# 编程指南

字符串（C# 编程指南）

如何：使用正则表达式搜索字符串（C# 编程指南）

LINQ and Strings

如何：使用正则表达式搜索字符串（C# 编程指南）

可以使用 System.Text.RegularExpressions.Regex 类搜索字符串。这些搜索有的非常简单，有的复杂到需要完全使用正则表达式。以下是使用 Regex 类搜索字符串的两个示例。有关更多信息，请参见.NET Framework 正则表达式。

以下代码是一个控制台应用程序，用于对数组中的字符串执行简单的不区分大小写的搜索。给定要搜索的字符串和包含搜索模式的字符串后，静态方法 Regex.IsMatch 将执行搜索。在本例中，使用第三个参数指示忽略大小写。有关更多信息，请参见System.Text.RegularExpressions.RegexOptions。

```
class TestRegularExpressions
{
    static void Main()
    {
        string[] sentences =
        {
            "C# code",
            "Chapter 2: Writing Code",
            "Unicode",
            "no match here"
        };

        string sPattern = "code";

        foreach (string s in sentences)
        {
            System.Console.Write("{0,24}", s);

            if (System.Text.RegularExpressions.Regex.IsMatch(s, sPa
            {
                System.Console.WriteLine("  (match for '{0}' found)
            }
            else
            {
                System.Console.WriteLine();
            }
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();

    }
}
/* Output:
           C# code  (match for 'code' found)
           Chapter 2: Writing Code  (match for 'code' found)
           Unicode  (match for 'code' found)
           no match here
*/
```

以下代码是一个控制台应用程序，此程序使用正则表达式验证数组中每个字符串的格式。验证要求每个字符串具有电话号码的形式，即用短划线将数字分成三组，前两组各包含三个数字，第三组包含四个数字。这是使用正则表达式 ^\d{3}-\d{3}-\d{4}$ 完成的。有关更多信息，请参见正则表达式语言 - 快速参考。

```
class TestRegularExpressionValidation
{
    static void Main()
    {
        string[] numbers =
        {
            "123-555-0190",
            "444-234-22450",
            "690-555-0178",
            "146-893-232",
            "146-555-0122",
            "4007-555-0111",
            "407-555-0111",
            "407-2-5555",
        };

        string sPattern = "^\\d{3}-\\d{3}-\\d{4}$";

        foreach (string s in numbers)
        {
            System.Console.Write("{0,14}", s);

            if (System.Text.RegularExpressions.Regex.IsMatch(s, sPa
            {
                System.Console.WriteLine(" - valid");
            }
            else
            {
                System.Console.WriteLine(" - invalid");
            }
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
      123-555-0190 - valid
     444-234-22450 - invalid
      690-555-0178 - valid
       146-893-232 - invalid
      146-555-0122 - valid
     4007-555-0111 - invalid
      407-555-0111 - valid
        407-2-5555 - invalid
*/
```

请参阅

System.Text.RegularExpressions.Regex

C# 编程指南

字符串（C# 编程指南）

.NET Framework 正则表达式

正则表达式语言 - 快速参考

如何：确定字符串是否表示数值（C# 编程指南）

若要确定字符串是否是指定数值类型的有效表示形式，请使用静态的 **TryParse** 方法，该方法由所有基元数值类型以及诸如 DateTime 和 IPAddress 这样的类型实现。下面的示例演示如何确定“108”是否为有效的 int 类型。

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

如果字符串包含非数值字符或者所包含的数值对于指定的特定类型而言太大或太小，**TryParse** 都将返回 false 并将 out 参数设置为零。否则，它将返回 true，并且将 out 参数设置为字符串的数值。

注意

有时，虽然字符串可能只包含数值字符，对于您使用的 **TryParse** 方法所属的类型却仍然会无效。例如，“256”对于 **byte** 类型不是有效值，但对于 **int** 类型有效。"98.6”对于 **int** 类型不是有效值，但对于 **decimal** 类型有效。

下面的示例演示如何对 **long**、**byte** 和 **decimal** 值的字符串表示形式使用 **TryParse**。

```
string numString = "1287543"; //"1287543.0" will return false for a
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
if (canConvert == true)
  Console.WriteLine("number1 now = {0}", number1);
else
  Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
canConvert = byte.TryParse(numString, out number2);
if (canConvert == true)
  Console.WriteLine("number2 now = {0}", number2);
else
  Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; //"27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
  Console.WriteLine("number3 now = {0}", number3);
else
  Console.WriteLine("number3 is not a valid decimal");
```

可靠编程

基元数值类型还实现了 **Parse** 静态方法，此方法会在字符串不是有效数字时引发异常。 **TryParse** 通常更加有效，因为它在数字无效时只是返回 false。

.NET Framework 安全性

请始终使用 **TryParse** 或 **Parse** 方法来验证用户在文本框和组合框等控件中输入的内容。

请参阅

如何：将字节数组转换为 int（C# 编程指南）

如何：将字符串转换为数字（C# 编程指南）

如何：在十六进制字符串与数值类型之间转换（C# 编程指南）

在 .NET Framework 中分析数值字符串

.NET Framework 中的格式化类型

如何：将字符串转换为 DateTime（C# 编程指南）

通常程序需要支持用户以字符串值的形式输入日期。若要将基于字符串的日期转换为 System.DateTime 对象，可以使用 Convert.ToDateTime(String) 方法或 DateTime.Parse(String) 静态方法，如下面的示例所示。

有关日期字符串的更多示例，请参见 Convert.ToDateTime(String)。

```
// Date strings are interpreted according to the current culture.
// If the culture is en-US, this is interpreted as "January 8, 2008
// but if the user's computer is fr-FR, this is interpreted as "Aug
string date = "01/08/2008";
DateTime dt = Convert.ToDateTime(date);
Console.WriteLine("Year: {0}, Month: {1}, Day: {2}", dt.Year, dt.Mo

// Specify exactly how to interpret the string.
IFormatProvider culture = new System.Globalization.CultureInfo("fr-

// Alternate choice: If the string has been input by an end user, y
// want to format it according to the current culture:
// IFormatProvider culture = System.Threading.Thread.CurrentThread.
DateTime dt2 = DateTime.Parse(date, culture, System.Globalization.D
Console.WriteLine("Year: {0}, Month: {1}, Day {2}", dt2.Year, dt2.M

/* Output (assuming first culture is en-US and second is fr-FR):
    Year: 2008, Month: 1, Day: 8
    Year: 2008, Month: 8, Day 1
 */
```

请参阅

字符串（C# 编程指南）

如何：在旧式编码与 Unicode 之间转换（C# 编程指南）

在 C# 中，内存中的所有字符串都是按 Unicode (UTF-16) 编码的。将数据从存储器移动到 **string** 对象中后，数据将自动转换为 UTF-16。如果数据仅包含从 0 到 127 的 ASCII 值，则此转换无需您执行任何额外的工作。但若源文本包含扩展的 ASCII 字节值（128 到 255），则默认情况下，将根据当前代码页解释扩展字符。若要指定应该根据其他某个代码页解释源文本，请使用 System.Text.Encoding 类，如下面的示例所示。

下面的示例演示如何转换按 8 位 ASCII 编码的文本文件，此转换根据 Windows 代码页 737 解释源文本。

```
class ANSIToUnicode
{
    static void Main()
    {
        // Create a file that contains the Greek work ψυχή (psyche
        // code page 737 ((DOS) Greek). You can also create the fi
        // to paste the characters into Microsoft Word and then "Sa
        // (Greek) encoding. (Word will actually create a six-byte
        System.IO.File.WriteAllBytes(@"greek.txt", new byte[] { 0xA

        // Specify the code page to correctly interpret byte values
        Encoding encoding = Encoding.GetEncoding(737); //(DOS) Gree
        byte[] codePageValues = System.IO.File.ReadAllBytes(@"greek

        // Same content is now encoded as UTF-16
        string unicodeValues = encoding.GetString(codePageValues);

        // Show that the text content is still intact in Unicode st
        // (Add a reference to System.Windows.Forms.dll)
        System.Windows.Forms.MessageBox.Show(unicodeValues);

        // Same content "ψυχή" is stored as UTF-8
        System.IO.File.WriteAllText(@"greek_unicode.txt", unicodeVa

        // Conversion is complete. Show the bytes to prove the conv
        Console.WriteLine("8-bit encoding byte values:");
        foreach(byte b in codePageValues)
            Console.Write("{0:X}-", b);

        Console.WriteLine();
        Console.WriteLine("Unicode values:");
        string unicodeString = System.IO.File.ReadAllText("greek_un
        System.Globalization.TextElementEnumerator enumerator =
            System.Globalization.StringInfo.GetTextElementEnumerato
        while(enumerator.MoveNext())
```

```
        {
            string s = enumerator.GetTextElement();
            int i = Char.ConvertToUtf32(s, 0);
            Console.Write("{0:X}-", i);
        }
        Console.WriteLine();

        // Keep the console window open in debug mode.
        Console.Write("Press any key to exit.");
        Console.ReadKey();
    }
    /*
     * Output:
        8-bit encoding byte values:
        AF-AC-AE-9E
        Unicode values:
        3C8-3C5-3C7-3B7
    */
}
```

请参阅

字符串（C# 编程指南）

如何：将 RTF 转换为纯文本（C# 编程指南）

RTF 格式是 Microsoft 于 20 世纪 80 年代后期开发的一种文件格式，旨在实现不同操作系统之间的文档交换。Microsoft Word 和 WordPad 都可以读写 RTF 文件。在 .NET Framework 中，可以使用 RichTextBox 控件创建支持 RTF 且支持用户以 WYSIWIG 方式将格式应用于文本的字处理器。

也可以使用 RichTextBox 以编程方式将 RTF 格式代码从文档中移除，从而将该文档转换为纯文本。执行这种类型的操作无需在 Windows 窗体中嵌入该控件。

在项目中使用 RichTextBox 控件

1. 添加对 System.Windows.Forms.dll 的引用。
2. 为 **System.Windows.Forms** 命名空间添加 using 指令（可选）。

下面的示例将示例 RTF 文件转换为纯文本。文件包含 RTF 格式设置 (如字体信息)，四个 Unicode 字符和四个扩展的 ASCII 字符。代码示例打开文件，将其内容设置为 RichTextBox，当 RTF，检索该内容作为文本，并在 MessageBox的文本和输出该文本到一个 UTF-8 格式的文件。

MessageBox 和输出文件包含以下文本：

```
The Greek word for "psyche" is spelled ψυχή. The Greek letters are
These characters are from the extended ASCII character set (Windows
```

```
// Use NotePad to save the following RTF code to a text file in the
// your .exe file for this project. Name the file test.rtf.
/*
{\rtf1\ansi\ansicpg1252\deff0\deflang1033{\fonttbl{\f0\fswiss\fchar
{\f1\fnil\fprq1\fcharset0 Courier New;}{\f2\fswiss\fprq2\fcharset0
{\colortbl ;\red0\green128\blue0;\red0\green0\blue0;}
{\*\generator Msftedit 5.41.21.2508;}
\viewkind4\uc1\pard\f0\fs20 The \i Greek \i0 word for "psyche" is s
These characters are from the extended \b ASCII \b0 character set (
*/
class ConvertFromRTF
{
    static void Main()
    {
        // If your RTF file isn't in the same folder as the .exe fi
        // specify the path to the file in the following assignment
        string path = @"test.rtf";

        //Create the RichTextBox. (Requires a reference to System.W
        System.Windows.Forms.RichTextBox rtBox = new System.Windows

        // Get the contents of the RTF file. When the contents of t
        // stored in the string (rtfText), the contents are encoded
        string rtfText = System.IO.File.ReadAllText(path);

        // Display the RTF text. This should look like the contents
        System.Windows.Forms.MessageBox.Show(rtfText);

        // Use the RichTextBox to convert the RTF code to plain tex
        rtBox.Rtf = rtfText;
        string plainText = rtBox.Text;

        // Display the plain text in a MessageBox because the conso
        // display the Greek letters. You should see the following
        //   The Greek word for "psyche" is spelled ψυχή. The Greek
        //   encoded in Unicode.
        //   These characters are from the extended ASCII character
        //   code page 1252): âäãå
        System.Windows.Forms.MessageBox.Show(plainText);

        // Output the plain text to a file, encoded as UTF-8\.
        System.IO.File.WriteAllText(@"output.txt", plainText);
    }
}
```

RTF 字符以八位编码。但是，除了从指定的代码页，扩展 ASCII 字符之外用户可以指定 Unicode 字符。由于 RichTextBox.Text 属性为 string 类型，因此字符按 Unicode UTF-16 进行编码。源 RTF 文档中的所有扩展的 ASCII 字符和 Unicode 字符都在文本输出中进行了正确的编码。

如果使用 File.WriteAllText 方法将文本写入磁盘，则文本将按 UTF-8（没有字节顺序标记）进行编码。

请参阅

System.Windows.Forms.RichTextBox

字符串（C# 编程指南）

类型（C# 编程指南）

类型、变量和值

C# 是一种强类型语言。每个变量和常量都有一个类型，每个计算为值的表达式也是如此。每个方法签名为每个输入参数和返回值指定一个类型。.NET Framework 类库定义了一组内置数值类型以及表示各种逻辑构造的更复杂的类型，例如文件系统、网络连接、对象的集合和数组及日期。典型 C# 程序使用类库中的类型，还使用为特定于该程序问题域的概念建模的用户定义类型。

类型中存储的信息可以包括：

- 该类型的变量所需的存储空间。
- 该类型可以表示的最大值和最小值。
- 该类型包含的成员（方法、字段、事件等）。
- 该类型所继承的基类型。
- 将在运行时为其分配变量内存的位置。
- 允许的运算种类。

编译器使用类型信息确保代码中执行的所有运算都是类型安全的。例如，如果声明了一个 int 类型的变量，则编译器允许您在加法和减法运算中使用此变量。如果尝试在一个 bool 类型的变量上执行相同的运算，则编译器会产生错误，如下面的示例所示：

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int'
int c = a + test;
```

注意

请 C 和 C++ 开发人员注意，在 C# 中，bool 不能转换为 int。

编译器将类型信息作为元数据嵌入到可执行文件中。公共语言运行时 (CLR) 会在运行时使用该元数据，以进一步确保它在分配和回收内存时类型安全。

在变量声明中指定类型

在程序中声明变量或常量时，必须指定其类型或者使用关键字 var 让编译器可以推断其类型。下面的示例演示了一些使用内置数值类型和复杂的用户定义类型的变量声明：

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

在方法签名中指定方法参数和返回值的类型。下面的签名中演示的方法需要用 int 作为输入参数并返回一个字符串：

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

声明了一个变量后，不能使用新类型重新对它进行声明，也不能向它赋与它的声明类型不兼容的值。例如，不能声明 int，然后向它赋予布尔值 true。但是，值可以转换为其他类型，例如将值赋给新变量或者作为方法参数传递时。编译器会自动执行不会导致数据丢失的类型转换。可能导致数据丢失的转换需要源代码内有强制转换。

有关详细信息，请参阅 强制转换和类型转换（C# 编程指南）。

内置类型

C# 提供了一组标准的内置数值类型，用来表示整数、浮点值、布尔表达式、文本字符、十进制值和其他数据类型。还有内置的 **string** 和 **object** 类型。您可以在任何 C# 程序中使用这些类型。有关内置类型的更多信息，请参见类型参考表（C# 参考）。

自定义类型

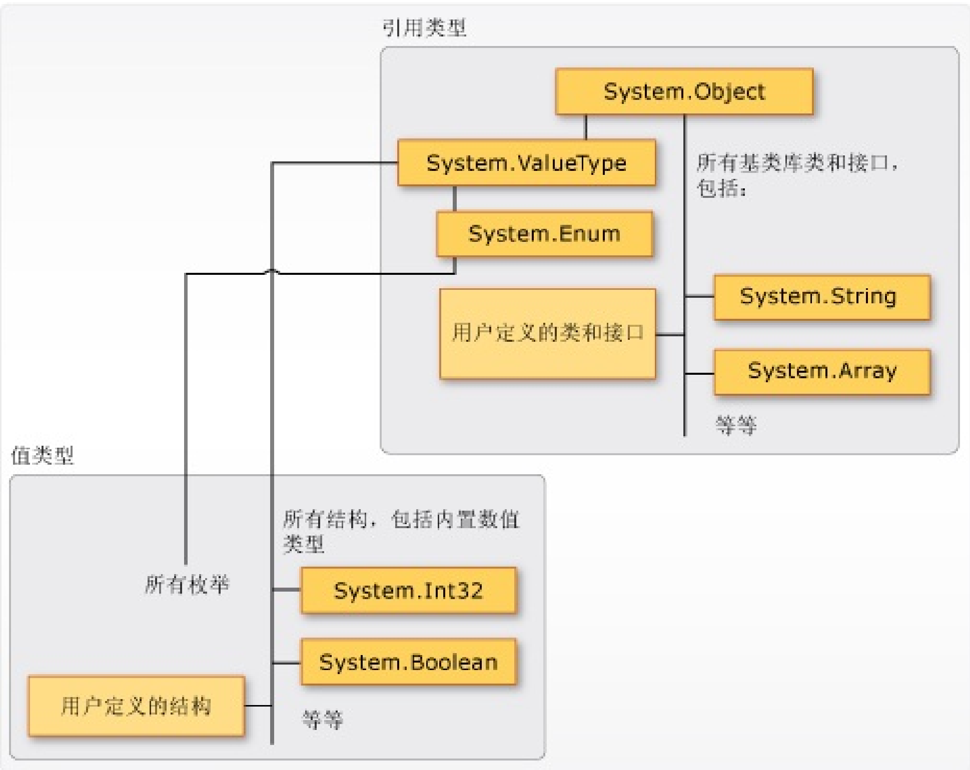
您可以使用 struct、class、interface 和 enum 结构创建自己的自定义类型。.NET Framework 类库本身是 Microsoft 提供的自定义类型的集合，您可以在自己的应用程序中使用它们。默认情况下，类库中最常用的类型在所有 C# 程序中均可用。而对于其他类型，则仅当您显式添加定义这些类型的程序集的项目引用后它们才可用。编译器拥有对该程序集的引用后，才可以在源代码中声明在该程序集中声明的类型的变量（和常量）。有关更多信息，请参见 .NET Framework Class Library（.NET Framework 类库）。

通用类型系统

请务必了解有关 .NET Framework 中的类型系统的以下两个基本点：

- 它支持继承原则。类型可从称为基类型的其他类型派生。派生类型继承基类型的方法、属性和其他成员（存在一些限制）。之后，基类型可从某些其他类型派生，这种情况下，派生类型继承其层次结构中这两个基类型的成员。包括如 System.Int32（C# 关键字：int）等内置数值类型在内的所有类型，最终都是从一个基类派生得到的，该基类即 System.Object （C# 关键字：object）。这种统一的类型层次结构称为常规类型系统 (CTS)。有关 C# 中的继承的更多信息，请参见继承（C# 编程指南）。
- CTS 中的每一个类型都被定义成了值类型或引用类型。这包括 .NET Framework 类库中的所有自定义类型以及您自己的用户定义类型。使用关键字 struct 定义的类型是值类型；所有内置数值类型都是 **structs**。使用关键字 class 定义的类型是引用类型。引用类型和值类型有不同的编译时规则和不同的运行时行为。

下图演示了 CTS 中的值类型和引用类型之间的关系。



CTS 中的值类型和引用类型

> 注意
>
> 可以看到，最常用的类型都组织到了 System 命名空间。但是，类型所在的命名空间与类型是值类型还是引用类型没有关系。

值类型

值类型是从派生自 System.Object 的 System.ValueType 派生的。派生自 System.ValueType 的类型在 CLR 中有特殊行为。值类型变量直接包含它们的值，这意味着内存在声明变量的任意上下文中都是以内联方式分配的。值类型变量没有单独的堆分配或垃圾回收开销。

值类型分为两个类别：struct 和 enum。

内置数值类型是结构，它们具有可以访问的属性和方法。

```
// Static method on type Byte.
byte b = Byte.MaxValue;
```

但是，您可以声明它们并向它们赋值，就如同它们是简单的非聚合类型一样：

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

例如，值类型是密封的，这意味着您不能从 System.Int32 派生类型，并且不能定义一个结构以便从任何用户定义的类或结构继承，因为结构只能从 System.ValueType 继承。但是，一个结构可以实现一个或更多个接口。可以将结构类型强制转换为接口类型；但这会使装箱操作在托管堆上的一个引用类型对象内包装该结构。在将值类型传递给将 System.Object 作为输入参数的方法时会发生装箱操作。有关详细信息，请参阅 装箱和取消装箱（C# 编程指南）。

可以使用 struct 关键字创建自己的自定义值类型。通常，结构可作为一小组相关变量的容器，如下面的示例所示：

```
public struct CoOrds
{
    public int x, y;

    public CoOrds(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

有关结构的更多信息，请参见结构（C# 编程指南）。有关 .NET Framework 中的值类型的更多信息，请参见常规类型系统。

另一类值类型是 enum。枚举定义一组已命名的整数常量。例如，.NET Framework 类库中的 System.IO.FileMode 枚举包含一组已命名的整数常量，用来指定应如何打开文件。其定义方式如下面的示例所示：

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

System.IO.FileMode.Create 常数的值为 2。但是，在阅读源代码时，名称可为人们提供更多信息，因此最好使用枚举而不使用常量文本数字。有关详细信息，请参阅 System.IO.FileMode。

所有枚举均是从继承自 System.ValueType 的 System.Enum 继承的。所有适用于结构的规则同样也适用于枚举。有关枚举的更多信息，请参见枚举类型（C# 编程指南）。

引用类型

定义为类、委托、数组或接口的类型是引用类型。 在运行时，当您声明引用类型的变量时，该变量会一直包含值 null，直至您使用 new 运算符显式创建对象的实例，或者为该变量分配已经在其他位置使用 new 创建的对象, 如下所示:

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

接口必须与实现它的类对象一起初始化。如果 MyClass 实现 IMyInterface，则您创建了 IMyInterface 的实例，如下面的示例所示：

```
IMyInterface iface = new MyClass();
```

创建对象时，将在托管堆上分配内存，变量只保存对对象位置的引用。对于托管堆上的类型，在 CLR 的自动内存管理功能（称为“垃圾回收”）对它们进行分配和回收时都需要系统开销。但是，也对垃圾回收进行了高度优化，在大多数情况下它不会引起性能问题。有关垃圾回收的更多信息，请参见自动内存管理。

所有数组都是引用类型，即使其元素是值类型也不例外。数组是从 System.Array 类隐式派生的，但可以通过 C# 提供的简化语法来声明和使用它们，如下面的示例所示：

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

引用类型完全支持继承。创建类时，可以从没有定义为 sealed 的任何其他接口或类继承，而其他类则可以从您创建的类继承并重写虚方法。有关如何创建自己的类的更多信息，请参见类和结构（C# 编程指南）。有关继承和虚方法的更多信息，请参见继承（C# 编程指南）。

文本值类型

在 C# 中，文本值从编译器接收类型。您可以通过在数字末尾追加一个字母来指定应如何类型化该数字文本。例如，若要指定应按浮点数来处理值 4.56，则在该数字后追加一个“f”或“F”：4.56f。如果没有追加字母，则编译器将为该文本推断一个类型。有关可以使用字母后缀指定的类型的更多信息，请参见值类型（C# 参考）中各个类型的参考页。

由于文本已类型化，且所有类型最终都是从 System.Object 派生，因此您可以编写和编译如下所示的代码：

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

泛型类型

一个类型可以通过一个或多个类型参数声明，而这些类型参数作为客户端代码在创建该类型的实例时提供的实际类型（具体类型）的占位符。这种类型称为"泛型类型"。例如，.NET Framework 类型 System.Collections.Generic.List<T> 有一个类型参数，按照约定该类型参数的名称为 T。在创建该类型的实例时，会指定列表将包含的对象的类型，例如字符串：

```
List<string> strings = new List<string>();
```

使用类型参数便可以重复使用相同的类存放任意类型的元素，而不必将每个元素都转换为对象。泛型集合类称为"强类型集合"，因为编译器知道集合中元素的特定类型，举例来说，如果尝试向上面示例中的 strings 对象添加整数，编译器会在编译时引发错误。有关详细信息，请参阅 泛型（C# 编程指南）。

隐式类型、匿名类型和可以为 null 的类型

如前面所述，您可以使用关键字 var 隐式类型化一个局部变量（非类成员）。该变量在编译时仍然会接收一个类型，但该类型是由编译器提供的。有关详细信息，请参阅 隐式类型的局部变量（C# 编程指南）。

某些情况下为相关值的简单集合创建命名类型是不方便的，因为这些相关值不准备在方法边界外存储或传递。可以创建匿名类型来实现此目的。有关更多信息，请参见匿名类型（C# 编程指南）。

普通的值类型不能有 null 值。但是，可以通过在类型后面附加 **?** 来创建可以为 null 值的类型。例如，**int?** 是一个也可以具有 null 值的 **int** 类型。在 CTS 中，可以为 null 的类型是泛型结构类型 System.Nullable<T> 的实例。在向其数值可能为 null 的数据库传入数据和从中传出数据时，可以为 null 的类型尤其有用。有关详细信息，请参阅 可以为 null 的类型（C# 编程指南）。

相关章节

有关详细信息，请参阅下列主题：

- 强制转换和类型转换（C# 编程指南）
- 装箱和取消装箱（C# 编程指南）
- 使用类型 dynamic（C# 编程指南）
- 值类型（C# 参考）
- 引用类型（C# 参考）
- 类和结构（C# 编程指南）

- 匿名类型（C# 编程指南）
- 泛型（C# 编程指南）
- Visual C# 2010 入门 中的变量与表达式

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

XML 数据类型的转换

整型表（C# 参考）

强制转换和类型转换（C# 编程指南）

由于 C# 是在编译时静态类型化的，因此变量在声明后就无法再次声明，或者无法用于存储其他类型的值，除非该类型可以转换为变量的类型。例如，不存在从整数到任意字符串的转换。因此，将 i 声明为整数后，就无法将字符串“Hello”赋予它，如下面的代码中所示。

```
int i;
i = "Hello"; // Error: "Cannot implicitly convert type 'string' to
```

但有时可能需要将值复制到其他类型的变量或方法参数中。例如，您可能有一个整数变量，您需要将该变量传递给参数类型化为 **double** 的方法。或者可能需要将类变量赋给接口类型的变量。这些类型的操作称为“类型转换”。在 C# 中，可以执行以下几种类型的转换：

- 隐式转换：由于该转换是一种安全类型的转换，不会导致数据丢失，因此不需要任何特殊的语法。例如，从较小整数类型到较大整数类型的转换以及从派生类到基类的转换都是这样的转换。
- 显式转换（强制转换）：显式转换需要强制转换运算符。在转换中可能丢失信息时或在出于其他原因转换可能不成功时，必须进行强制转换。典型的例子包括从数值到精度较低或范围较小的类型的转换和从基类实例到派生类的转换。
- 用户定义的转换：可以定义一些特殊的方法来执行用户定义的转换，从而使不具有基类–派生类关系的自定义类型之间可以显式和隐式转换。有关更多信息，请参见 转换运算符（C# 编程指南）。
- 使用帮助程序类的转换：若要在不兼容的类型之间进行转换，例如在整数与 System.DateTime 对象之间转换，或者在十六进制字符串与字节数组之间转换，则可以使用 System.BitConverter 类、System.Convert 类和内置数值类型的 **Parse** 方法，例如 Int32.Parse。有关更多信息，请参见如何：将字节数组转换为 int（C# 编程指南）、如何：将字符串转换为数字（C# 编程指南）和如何：在十六进制字符串与数值类型之间转换（C# 编程指南）。

隐式转换

对于内置数值类型，如果要存储的值无需截断或四舍五入即可适应变量，则可以进行隐式转换。例如，long 类型的变量（8 字节整数）能够存储 int（在 32 位计算机上为 4 字节）可存储的任何值。在下面的示例中，编译器先将右侧的值隐式转换为 **long** 类型，再将它赋给 bigNum。

```
// Implicit conversion. num long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

有关所有隐式数值转换的完整列表，请参见隐式数值转换表（C# 参考）。

对于引用类型，隐式转换始终存在于从一个类转换为该类的任何一个直接或间接的基类或接口的情况。由于派生类始终包含基类的所有成员，因此不必使用任何特殊语法。

```
Derived d = new Derived();
Base b = d; // Always OK.
```

显式转换

但是，如果进行转换可能会导致信息丢失，则编译器会要求执行显式转换，显式转换也称为“强制转换”。强制转换是显式通知编译器您打算进行转换且您知道可能会发生数据丢失的一种方式。若要执行强制转换，请在要转换的值或变量前面的圆括号中指定要强制转换到的类型。下面的程序将 double 强制转换为 int。如不强制转换则该程序不会进行编译。

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

有关支持的显式数值转换的列表，请参见显式数值转换表（C# 参考）。

对于引用类型，如果需要从基类型转换为派生类型，则必须进行显式强制转换：

```
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe) a;
```

引用类型之间的强制转换操作不会更改基础对象的运行时类型；它只更改用作对该对象的引用的值的类型。有关更多信息，请参见 多态性（C# 编程指南）。

运行时的类型转换异常

在某些引用类型转换中，编译器无法确定强制转换是否会有效。正确进行编译的强制转换操作有可能在运行时失败。如下面的示例所示，类型强制转换在运行时失败将导致引发 InvalidCastException。

```
using System;

class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // Cause InvalidCastException at run time
        // because Mammal is not convertible to Reptile.
        Reptile r = (Reptile)a;
    }

}
```

C# 提供 is 和 as 运算符，使您可以在实际执行强制转换之前测试兼容性。有关更多信息，请参见 如何：使用 as 和 is 运算符安全地进行强制转换（C# 编程指南）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

重要章节

More About Variables 在 Beginning Visual C# 2010

请参阅

C# 编程指南

类型（C# 编程指南）

() 运算符（C# 参考）

explicit（C# 参考）

implicit（C# 参考）

转换运算符（C# 编程指南）

Generalized Type Conversion

Exported Type Conversion

如何：将字符串转换为数字（C# 编程指南）

装箱和取消装箱（C# 编程指南）

装箱是将值类型转换为 **object** 类型或由此值类型实现的任何接口类型的过程。当 CLR 对值类型进行装箱时，会将该值包装到 System.Object 内部，再将后者存储在托管堆上。取消装箱将从对象中提取值类型。装箱是隐式的；取消装箱是显式的。装箱和取消装箱的概念是类型系统 C# 统一视图的基础，其中任一类型的值都被视为一个对象。

在下面的示例中，将整型变量 i 进行了装箱并分配给对象 o。

```
int i = 123;
// The following line boxes i.
object o = i;
```

然后，可以将对象 o 取消装箱并分配给整型变量 i：

```
o = 123;
i = (int)o;  // unboxing
```

以下示例演示如何在 C# 中使用装箱。

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));

// List example.
// Create a list of objects to hold a heterogeneous collection
// of elements.
List<object> mixedList = new List<object>();

// Add a string element to the list.
mixedList.Add("First Group:");

// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
```

```
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j]);

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

// The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
Console.WriteLine("Sum: " + sum);

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
// Second Group:
// 5
// 6
// 7
// 8
// 9
// Sum: 30
```

性能

相对于简单的赋值而言，装箱和取消装箱过程需要进行大量的计算。对值类型进行装箱时，必须分配并构造一个新对象。取消装箱所需的强制转换也需要进行大量的计算，只是程度较轻。有关更多信息，请参见性能。

装箱

装箱用于在垃圾回收堆中存储值类型。装箱是值类型到 **object** 类型或到此值类型所实现的任何接口类型的隐式转换。对值类型装箱会在堆中分配一个对象实例，并将该值复制到新的对象中。

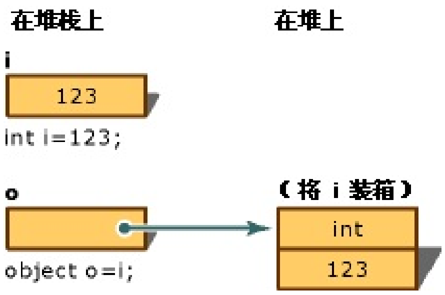
请看以下值类型变量的声明：

```
int i = 123;
```

以下语句对变量 i 隐式应用了装箱操作：

```
// Boxing copies the value of i into object o.
object o = i;
```

此语句的结果是在堆栈上创建对象引用 o，而在堆上则引用 **int** 类型的值。该值是赋给变量 i 的值类型值的一个副本。下图说明了两个变量 i 和 o 之间的差异。



装箱转换

还可以像下面的示例一样执行显式装箱，但显式装箱从来不是必需的：

```
int i = 123;
object o = (object)i;  // explicit boxing
```

说明

此示例使用装箱将整型变量 i 转换为对象 o。这样一来，存储在变量 i 中的值就从 123 更改为 456。该示例表明原始值类型和装箱的对象使用不同的内存位置，因此能够存储不同的值。

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

        // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i does not effect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
    The value-type value = 456
    The object-type value = 123
*/
```

取消装箱

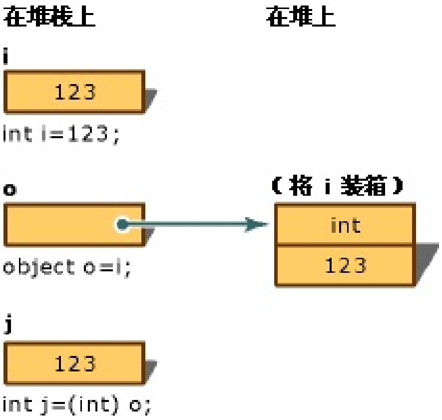
取消装箱是从 **object** 类型到值类型或从接口类型到实现该接口的值类型的显式转换。取消装箱操作包括：

- 检查对象实例，以确保它是给定值类型的装箱值。
- 将该值从实例复制到值类型变量中。

下面的语句演示装箱和取消装箱两种操作：

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;   // unboxing
```

下图演示上述语句的结果。



取消装箱转换

要在运行时成功取消装箱值类型，被取消装箱的项必须是对一个对象的引用，该对象是先前通过装箱该值类型的实例创建的。尝试取消装箱 **null** 会导致 NullReferenceException。尝试取消装箱对不兼容值类型的引用会导致 InvalidCastException。

下面的示例演示无效的取消装箱及引发的 **InvalidCastException**。使用 **try** 和 **catch**，在发生错误时显示错误信息。

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i;  // implicit boxing

        try
        {
            int j = (short)o;  // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Error: Incorrect unboxing
        }
    }
}
```

此程序输出：

Specified cast is not valid.Error: Incorrect unboxing.

如果将下列语句：

```
int j = (short) o;
```

更改为：

```
int j = (int) o;
```

将执行转换，并将得到以下输出：

Unboxing OK.

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

相关章节

更多相关信息：

- 引用类型
- 值类型

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

使用类型 dynamic（C# 编程指南）

Visual C# 2010 引入了一个新类型 **dynamic**。该类型是一种静态类型，但类型为 **dynamic** 的对象会跳过静态类型检查。大多数情况下，该对象就像具有类型 **object** 一样。在编译时，将假定类型化为 **dynamic** 的元素支持任何操作。因此，您不必考虑对象是从 COM API、从动态语言（例如 IronPython）、从 HTML 文档对象模型 (DOM)、从反射还是从程序中的其他位置获取自己的值。但是，如果代码无效，则在运行时会捕获到错误。

例如，如果以下代码中的实例方法 exampleMethod1 只有一个形参，则编译器会将对该方法的第一个调用 ec.exampleMethod1(10, 4) 识别为无效，原因是它包含两个实参。该调用将导致编译器错误。编译器不会检查对该方法的第二个调用 dynamic_ec.exampleMethod1(10, 4)，原因是 dynamic_ec 的类型为 **dynamic**。因此，不会报告编译器错误。但是，该错误不会被无限期疏忽。它将在运行时被捕获，并导致运行时异常。

```
  static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, wheth
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

```
  class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

在这些示例中，编译器的作用是将有关每个语句的预期作用的信息一起打包到类型化为 **dynamic** 的对象或表达式。在运行时，将对存储的信息进行检查，并且任何无效的语句将会导致运行时异常。

大多数动态操作的结果是其本身 **dynamic**。例如，如果将鼠标指针放在以下示例中使用的 testSum 上，则 IntelliSense 将显示类型 **“(局部变量)dynamic testSum”**。

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

结果不是 **dynamic** 的操作包括从 **dynamic** 到另一种类型的转换，以及包括类型为 **dynamic** 的参数的构造函数调用。例如，以下声明中 testInstance 的类型为 ExampleClass，而不是 **dynamic**。

```
var testInstance = new ExampleClass(d);
```

下一节“转换”中显示了转换示例。

转换

动态对象和其他类型之间的转换非常简单。这样，开发人员将能够在动态行为和非动态行为之间切换。

任何对象都可隐式转换为动态类型，如下面的示例所示。

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

反之，隐式转换也可动态地应用于类型为 **dynamic** 的任何表达式。

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

使用类型为 dynamic 的参数重载决策

如果方法调用中的一个或多个参数具有类型 **dynamic**，或者方法调用的接收方的类型为 **dynamic**，则会在运行时（而不是在编译时）进行重载决策。在下面的示例中，如果唯一可访问的 exampleMethod2 方法定义为接受字符串参数，则将 d1 作为参数发送不会导致编译器错误，但却会导致运行时异常。重载决策之所以会在运行时失败，原因是 d1 的运行时类型为 **int**，而 exampleMethod2 需要字符串。

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even th
// dynamic. A run-time exception is raised because the run-time typ
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

动态语言运行时

动态语言运行时 (DLR) 是 .NET Framework 4 中的一个新 API。它提供了支持 C# 中的 **dynamic** 类型的基础结构，还提供了诸如 IronPython 和 IronRuby 等动态编程语言的实现。有关 DLR 的更多信息，请参见动态语言运行时概述。

COM 互操作

Visual C# 2010 包括若干功能，这些功能改善了与 COM API（例如 Office 自动化 API）的互操作体验。这些改进之处包括 **dynamic** 类型以及命名参数和可选参数的使用。

通过将类型指定为 **object**，许多 COM 方法都允许参数类型和返回类型发生变化。这样，就必须显式强制转换值，以便与 C# 中的强类型变量保持协调。如果使用 /link (C# Compiler Options) 选项进行编译，则 **dynamic** 类型的引入使您能够将 COM 签名中出现的 **object** 实例看作 **dynamic** 类型，从而避免了大量的强制转换。例如，下面的语句对比了在使用 **dynamic** 类型和不使用 **dynamic** 类型的情况下如何访问 Microsoft Office Excel 电子表格中的单元格。

```
// Before the introduction of dynamic.
((Excel.Range)excelApp.Cells[1, 1]).Value2 = "Name";
Excel.Range range2008 = (Excel.Range)excelApp.Cells[1, 1];
```

```
// After the introduction of dynamic, the access to the Value pro
// the conversion to Excel.Range are handled by the run-time COM bi
excelApp.Cells[1, 1].Value = "Name";
Excel.Range range2010 = excelApp.Cells[1, 1];
```

相关主题

标题	描述
dynamic（C# 参考）	描述 **dynamic** 关键字的用法。
动态语言运行时概述	提供有关 DLR 的概述，DLR 是一种运行时环境，它将一组适用于动态语言的服务添加到公共语言运行时 (CLR)。
演练：创建和使用动态对象（C# 和 Visual Basic）	提供了有关如何创建自定义动态对象以及创建访问 **IronPython** 库的对象的分步说明。
如何：通过使用 Visual C# 功能访问 Office 互操作对象（C# 编程指南）	演示如何创建一个项目，该项目使用了命名参数和可选参数、**dynamic** 类型以及可简化对 Office API 对象的访问的其他增强功能。

如何：使用 as 和 is 运算符安全地进行强制转换（C# 编程指南）

由于对象是多态的，因此基类类型的变量可以保存派生类型。若要访问派生类型的方法，需要将值强制转换回该派生类型。不过，在这些情况下，如果只尝试进行简单的强制转换，会导致引发 InvalidCastException 的风险。这就是 C# 提供 is 和 as 运算符的原因。您可以使用这两个运算符来测试强制转换是否会成功，而没有引发异常的风险。通常，**as** 运算符更高效一些，因为如果可以成功进行强制转换，它会实际返回强制转换值。而 **is** 运算符只返回一个布尔值。因此，如果只想确定对象的类型，而无需对它进行实际强制转换，则可以使用 is 运算符。

下面的示例演示如何使用 **is** 和 **as** 运算符从一个引用类型强制转换为另一个引用类型，而没有引发异常的风险。此示例还演示如何对 **as** 运算符使用可以为 null 值的类型。

```
class SafeCasting
{
    class Animal
    {
        public void Eat() { Console.WriteLine("Eating."); }
        public override string ToString()
        {
            return "I am an animal.";
        }
    }
    class Mammal : Animal { }
    class Giraffe : Mammal { }

    class SuperNova { }

    static void Main()
    {
        SafeCasting app = new SafeCasting();

        // Use the is operator to verify the type.
        // before performing a cast.
        Giraffe g = new Giraffe();
        app.UseIsOperator(g);

        // Use the as operator and test for null
        // before referencing the variable.
        app.UseAsOperator(g);

        // Use the as operator to test
        // an incompatible type.
        SuperNova sn = new SuperNova();
        app.UseAsOperator(sn);
```

```
        // Use the as operator with a value type.
        // Note the implicit conversion to int? in
        // the method body.
        int i = 5;
        app.UseAsWithNullable(i);

        double d = 9.78654;
        app.UseAsWithNullable(d);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    void UseIsOperator(Animal a)
    {
        if (a is Mammal)
        {
            Mammal m = (Mammal)a;
            m.Eat();
        }
    }

    void UseAsOperator(object o)
    {
        Mammal m = o as Mammal;
        if (m != null)
        {
            Console.WriteLine(m.ToString());
        }
        else
        {
            Console.WriteLine("{0} is not a Mammal", o.GetType().Na
        }
    }

    void UseAsWithNullable(System.ValueType val)
    {
        int? j = val as int?;
        if (j != null)
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("Could not convert " + val.ToString()
        }
    }
}
```

请参阅

类型（C# 编程指南）

强制转换和类型转换（C# 编程指南）

可以为 null 的类型（C# 编程指南）

如何：将字节数组转换为 int（C# 编程指南）

此示例演示如何使用 BitConverter 类将字节数组转换为 int 并返回字节数组。例如，在从网络读取字节之后，可能需要将字节转换为内置数据类型。除了示例中的 ToInt32(Byte[], Int32) 方法之外，下表列出了 BitConverter 类中将字节（来自字节数组）转换为其他内置类型的方法。

返回类型	方法
bool	ToBoolean(Byte[], Int32)
char	ToChar(Byte[], Int32)
double	ToDouble(Byte[], Int32)
short	ToInt16(Byte[], Int32)
int	ToInt32(Byte[], Int32)
long	ToInt64(Byte[], Int32)
float	ToSingle(Byte[], Int32)
ushort	ToUInt16(Byte[], Int32)
uint	ToUInt32(Byte[], Int32)
ulong	ToUInt64(Byte[], Int32)

此示例实例化字节数组，并在计算机结构为 little-endian 的情况下反转数组（即首先存储最低有效字节），然后调用 ToInt32(Byte[], Int32) 方法以将数组中的四个字节转换为 **int**。 ToInt32(Byte[], Int32) 的第二个参数指定字节数组的起始索引。

注意
输出可能会根据计算机结构的 endian 设置而不同。

```
byte[] bytes = { 0, 0, 0, 25 };

// If the system architecture is little-endian (that is, little enc
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine("int: {0}", i);
// Output: int: 25
```

在此示例中，调用 BitConverter 类的 GetBytes(Int32) 方法以将 **int** 转换为字节数组。

注意
输出可能会根据计算机结构的 endian 设置而不同。

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

请参阅

BitConverter

IsLittleEndian

类型（C# 编程指南）

如何：将字符串转换为数字（C# 编程指南）

可以使用 Convert 类中的方法或使用各种数值类型（int、long、float 等）中的 TryParse 方法将字符串转换为数字。

如果你具有字符串，则调用 TryParse 方法（例如 int.TryParse(“11”)）会稍微更加高效且简单。使用 Convert 方法对于实现 IConvertible 的常规对象更有用。

可以对预期字符串会包含的数值类型（如 System.Int32 类型）使用 Parse 或 TryParse 方法。 Convert.ToUInt32 方法在内部使用 Parse。如果字符串的格式无效，则 Parse 会引发异常，而 TryParse 会返回 false。

Parse 和 TryParse 方法会忽略字符串开头和末尾的空格，但所有其他字符必须是组成合适数值类型（int、long、ulong、float、decimal 等）的字符。组成数字的字符中的任何空格都会导致错误。例如，可以使用 decimal.TryParse 分析“10”、“10.3”、“ 10 ”，但不能使用此方法分析从“10X”、“1 0”（注意空格）、“10 .3”（注意空格）、“10e1”（float.TryParse 在此处适用）等中分析出 10。

下面的示例演示了对 Parse 和 TryParse 的成功调用和不成功的调用。

```
using System;
using System.Linq;
using System.Collections;
using System.Collections.Generic;
```

```
int numVal = Int32.Parse("-105");
Console.WriteLine(numVal);
// Output: -105
```

```
// TryParse returns true if the conversion succeeded
// and stores the result in j.
int j;
if (Int32.TryParse("-105", out j))
    Console.WriteLine(j);
else
    Console.WriteLine("String could not be parsed.");
// Output: -105
```

```
try
{
    int m = Int32.Parse("abc");
}
catch (FormatException e)
{
    Console.WriteLine(e.Message);
}
// Output: Input string was not in a correct format.
```

```
string inputString = "abc";
int numValue;
bool parsed = Int32.TryParse(inputString, out numValue);

if (!parsed)
    Console.WriteLine("Int32.TryParse could not parse '{0}' to an i

// Output: Int32.TryParse could not parse 'abc' to an int.
```

```
// This snippet shows a couple of examples that extract number char
// beginning of the string to avoid TryParse errors.
StringBuilder sb = new StringBuilder();
var str = "  10FFxxx";
foreach (char c in str) {
    // Check for numeric characters (hex in this case).  Add "." ar
    // and remove letters.  Include initial space because it is har
    if ((c >= '0' && c <= '9') || (c >= 'A' && c <= 'F') || (c >= 
        sb.Append(c);
    }
    else
        break;
}
if (int.TryParse(sb.ToString(), System.Globalization.NumberStyles.H
    Console.WriteLine(sb.ToString());

str = "   -10FFXXX";
sb.Clear();
foreach (char c in str) {
    // Check for numeric characters (allow negative in this case bu
    // Though we use int.TryParse in the previous example and this 
    // allow a sign character (-) AND hex digits at the same time.
    // Include initial space because it is harmless.
    if ((c >= '0' && c <= '9') || c == ' ' || c == '-') {
        sb.Append(c);
    } else
        break;
}
if (int.TryParse(sb.ToString(), out i))
    Console.WriteLine(sb.ToString());
```

下表列出了 Convert 类中可使用的一些方法。

数值类型	方法
decimal	ToDecimal(String)
float	ToSingle(String)
double	ToDouble(String)
short	ToInt16(String)
int	ToInt32(String)
long	ToInt64(String)
ushort	ToUInt16(String)
uint	ToUInt32(String)
ulong	ToUInt64(String)

此示例调用 Convert.ToInt32(String) 方法将输入的 string 转换为 int。代码将捕获此方法可能引发的最常见的两个异常：FormatException 和 OverflowException。如果该数字可以递增而不溢出整数存储位置，则程序使结果加上 1 并打印输出。

```
using System;
using System.Linq;
using System.Collections;
using System.Collections.Generic;
```

```
static void Main(string[] args)
{
    int numVal = -1;
    bool repeat = true;

    while (repeat)
    {
        Console.WriteLine("Enter a number between −2,147,483,648 ar

        string input = Console.ReadLine();

        // ToInt32 can throw FormatException or OverflowException.
        try
        {
            numVal = Convert.ToInt32(input);
        }
        catch (FormatException e)
        {
            Console.WriteLine("Input string is not a sequence of di
        }
        catch (OverflowException e)
        {
            Console.WriteLine("The number cannot fit in an Int32.")
        }
        finally
        {
            if (numVal < Int32.MaxValue)
            {
                Console.WriteLine("The new value is {0}", numVal +
            }
            else
            {
                Console.WriteLine("numVal cannot be incremented bey
            }
        }
        Console.WriteLine("Go again? Y/N");
        string go = Console.ReadLine();
        if (go == "Y" || go == "y")
        {
            repeat = true;
        }
```

```
            else
            {
                repeat = false;
            }
        }
        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
    // Sample Output:
    // Enter a number between -2,147,483,648 and +2,147,483,647 (inclus
    // 473
    // The new value is 474
    // Go again? Y/N
    // y
    // Enter a number between -2,147,483,648 and +2,147,483,647 (inclus
    // 2147483647
    // numVal cannot be incremented beyond its current value
    // Go again? Y/N
    // Y
    // Enter a number between -2,147,483,648 and +2,147,483,647 (inclus
    // -1000
    // The new value is -999
    // Go again? Y/N
    // n
    // Press any key to exit.
```

请参阅

类型（C# 编程指南）

如何：确定字符串是否表示数值（C# 编程指南）

.NET Framework 4 格式设置实用工具

如何：在十六进制字符串与数值类型之间转换（C# 编程指南）

以下示例演示如何执行下列任务：

- 获取字符串中每个字符的十六进制值。
- 获取与十六进制字符串中的每个值对应的字符。
- 将十六进制 **string** 转换为整型。
- 将十六进制 **string** 转换为浮点型。
- 将字节数组转换为十六进制 **string**。

此示例输出 **string** 中的每个字符的十六进制值。首先，它将 **string** 分析为字符数组，然后对每个字符调用 ToInt32(Char) 以获取相应的数字值。最后，在 **string** 中将数字的格式设置为十六进制表示形式。

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the decimal value to a hexadecimal value in string f
    string hexOutput = String.Format("{0:X}", value);
    Console.WriteLine("Hexadecimal value of {0} is {1}", letter, he
}
/* Output:
   Hexadecimal value of H is 48
    Hexadecimal value of e is 65
    Hexadecimal value of l is 6C
    Hexadecimal value of l is 6C
    Hexadecimal value of o is 6F
    Hexadecimal value of   is 20
    Hexadecimal value of W is 57
    Hexadecimal value of o is 6F
    Hexadecimal value of r is 72
    Hexadecimal value of l is 6C
    Hexadecimal value of d is 64
    Hexadecimal value of ! is 21
 */
```

此示例分析十六进制值的 **string** 并输出对应于每个十六进制值的字符。首先，它调用 Split(Char[]) 方法以获取每个十六进制值作为数组中的单个 **string**。然后调用 ToInt32(String, Int32) 以将十六进制转换为表示为 int 的十进制值。示例中演示了

用于获取对应于该字符代码的字符的两种不同方法。第一种方法是使用 ConvertFromUtf32(Int32)，它将对应于整型参数的字符作为 **string** 返回。第二种方法是将 **int** 显式转换为 char。

```
string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (String hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, ch
                        hex, value, stringValue, charValue);
}
/* Output:
    hexadecimal value = 48, int value = 72, char value = H or H
    hexadecimal value = 65, int value = 101, char value = e or e
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 20, int value = 32, char value =   or
    hexadecimal value = 57, int value = 87, char value = W or W
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 72, int value = 114, char value = r or r
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 64, int value = 100, char value = d or d
    hexadecimal value = 21, int value = 33, char value = ! or !
*/
```

此示例演示了将十六进制 **string** 转换为整数的另一种方法，即调用 Parse(String, NumberStyles) 方法。

```
string hexString = "8E2";
int num = Int32.Parse(hexString, System.Globalization.NumberStyles
Console.WriteLine(num);
//Output: 2274
```

下面的示例演示如何使用 System.BitConverter 类和 Int32.Parse 方法将十六进制 **string** 转换为浮点型。

```
string hexString = "43480170";
uint num = uint.Parse(hexString, System.Globalization.NumberStyles.

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine("float convert = {0}", f);

// Output: 200.0056
```

下面的示例演示如何使用 System.BitConverter 类将字节数组转换为十六进制字符串。

```
byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
  01-AA-B1-DC-10-DD
  01AAB1DC10DD
 */
```

请参阅

标准数字格式字符串

类型（C# 编程指南）

如何：确定字符串是否表示数值（C# 编程指南）

不安全代码和指针（C# 编程指南）

为了保持类型安全，默认情况下，C# 不支持指针算法。不过，通过使用 unsafe 关键字，可以定义可使用指针的不安全上下文。有关指针的更多信息，请参见主题指针类型。

> 注意
>
> 在公共语言运行时 (CLR) 中，不安全代码是指无法验证的代码。C# 中的不安全代码不一定是危险的；只是其安全性无法由 CLR 进行验证的代码。因此，CLR 只对在完全受信任的程序集中的不安全代码执行操作。如果使用不安全代码，由您负责确保您的代码不会引起安全风险或指针错误。

不安全代码概述

不安全代码具有下列属性：

- 方法、类型和可被定义为不安全的代码块。
- 在某些情况下，通过移除数组界限检查，不安全代码可提高应用程序的性能。
- 当调用需要指针的本机函数时，需要使用不安全代码。
- 使用不安全代码将引起安全风险和稳定性风险。
- 在 C# 中，为了编译不安全代码，必须用 /unsafe 编译应用程序。

相关章节

有关更多信息，请参见：

- 指针类型（C# 编程指南）
- 固定大小的缓冲区（C# 编程指南）
- 如何：使用指针复制字节数组（C# 编程指南）
- unsafe（C# 参考）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

固定大小的缓冲区（C# 编程指南）

在 C# 中，可以使用 fixed 语句在数据结构中创建带有固定大小数组的缓冲区。使用现有代码（如使用其他语言、预先存在的 DLL 或 COM 项目编写的代码）时，这种方法非常有用。固定数组可采用允许普通结构成员使用的任何特性或修饰符。唯一的限制是，数组类型必须是 **bool**、**byte**、 **char**、**short**、**int**、**long**、**sbyte**、**ushort**、**uint**、**ulong**、**float** 或 **double**。

```
private fixed char name[30];
```

备注

在早期版本的 C# 中，声明 C++ 样式的固定大小结构是很困难的，因为包含数组的 C# 结构不包含数组元素。相反，该结构包含对元素的引用。

C# 2.0 添加了在 struct（当用在 unsafe 代码块中时）中嵌入固定大小的数组的功能。

例如，在 C# 2.0 之前，下面的 **struct** 的大小为 8 字节。 pathName 数组是对堆分配数组的引用：

```
public struct MyArray
{
       public char[] pathName;
       private int reserved;
}
```

从 C# 2.0 开始，**struct** 可以包含嵌入的数组。在下面的示例中，fixedBuffer 数组有固定的大小。若要访问数组的元素，应使用 **fixed** 语句建立指向第一个元素的指针。 **fixed** 语句将 fixedBuffer 实例固定到内存中的特定位置。

```
namespace FixedSizeBuffers
{
    internal unsafe struct MyBuffer
    {
        public fixed char fixedBuffer[128];
    }

    internal unsafe class MyClass
    {
        public MyBuffer myBuffer = default(MyBuffer);
    }

    internal class Program
    {
        static void Main()
        {
            MyClass myC = new MyClass();

            unsafe
            {
                // Pin the buffer to a fixed location in memory.
                fixed (char* charPtr = myC.myBuffer.fixedBuffer)
                {
                    *charPtr = 'A';
                }
            }
        }
    }
}
```

128 个元素的 **char** 数组的大小为 256 字节。在固定大小的 char 缓冲区中，每个字符始终占用两个字节，而与编码无关。即使将 char 缓冲区封送到具有 CharSet = CharSet.Auto 或 CharSet = CharSet.Ansi 的 API 方法或结构，也是如此。有关更多信息，请参见 CharSet。

另一种常见的固定大小的数组是 bool 数组。 **bool** 数组中元素的大小始终为一个字节。 **bool** 数组不适合于创建位数组或缓冲区。

> 注意
>
> 除了用 stackalloc 创建的内存之外，C# 编译器和公共语言运行时 (CLR) 不执行任何安全缓冲区溢出检查。与所有不安全代码一样，请谨慎使用。

不安全缓冲区与常规数组在以下方面不同：

- 不安全缓冲区只能用在不安全上下文中。
- 不安全缓冲区始终是向量（或一维数组）。
- 数组的声明应包括计数，如 char id[8]。而不能使用 char id[]。

- 不安全缓冲区只能是不安全上下文中的结构的实例字段。

请参阅

C# 编程指南

不安全代码和指针（C# 编程指南）

fixed 语句（C# 参考）

互操作性（C# 编程指南）

指针类型（C# 编程指南）

在不安全的上下文中，类型可以是指针类型、值类型或引用类型。指针类型声明采用下列形式之一：

```
type* identifier;
void* identifier; //allowed but not recommended
```

以下任一类型均可为指针类型：

- sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal 或 bool。
- 任何枚举类型。
- 任何指针类型。
- 任何仅包含非托管类型字段的用户定义的结构类型。

指针类型不从 object 继承，并且指针类型与 **object** 之间不存在转换。此外，装箱和取消装箱不支持指针。但是，你可在不同的指针类型之间以及指针类型和整型之间进行转换。

在同一个声明中声明多个指针时，星号 (*) 仅与基础类型一起写入；而不是用作每个指针名称的前缀。例如：

```
int* p1, p2, p3;   // Ok
int *p1, *p2, *p3;   // Invalid in C#
```

指针不能指向引用或包含引用的 struct，因为无法对对象引用进行垃圾回收，即使有指针指向它也是如此。垃圾回收器并不跟踪是否有任何类型的指针指向对象。

myType* 类型的指针变量的值为 myType 类型的变量的地址。下面是指针类型声明的示例：

示例	说明
int* p	p 是指向整数的指针。
int** p	p 是指向整数的指针的指针。
int*[] p	p 是指向整数的指针的一维数组。
char* p	p 是指向字符的指针。
void* p	p 是指向未知类型的指针。

指针间接寻址运算符 * 可用于访问位于指针变量所指向的位置的内容。例如，请考虑以下声明：

```
int* myVariable;
```

表达式 *myVariable* 表示在 *myVariable* 中包含的地址处找到的 **int* 变量。

主题fixed 语句（C# 参考）和指针转换（C# 编程指南）包含了多个指针示例。下面的示例显示需要 **unsafe** 关键字和 **fixed** 语句以及如何递增内部指针。你可将此代码粘贴到控制台应用程序的 Main 函数中来运行它。（记住在“项目设计器”中启用不安全代码；选择菜单栏上的“项目”、“属性”，然后选择“生成”选项卡中的“允许不安全代码”。）

```
// Normal pointer to an object.
int[] a = new int[5] {10, 20, 30, 40, 50};
// Must be in unsafe code to use interior pointers.
unsafe
{
    // Must pin object on heap so that it doesn't move while using
    fixed (int* p = &a[0])
    {
        // p is pinned as well as object, so create another pointe
        int* p2 = p;
        Console.WriteLine(*p2);
        // Incrementing p2 bumps the pointer by four bytes due to i
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("--------");
        Console.WriteLine(*p);
        // Deferencing p and incrementing changes the value of a[0]
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}

Console.WriteLine("--------");
Console.WriteLine(a[0]);
Console.ReadLine();

// Output:
//10
//20
//30
//--------
//10
//11
//12
//--------
//12
```

你无法对 void* 类型的指针应用间接寻址运算符。但是，你可以使用强制转换将 void 指针转换为任何其他指针类型，反之亦然。

指针可以为 **null**。将间接寻址运算符应用于 null 指针将导致由实现定义的行为。

请注意，在方法之间传递指针会导致未定义的行为。示例通过 Out 或 Ref 参数或作为函数结果返回一个指向局部变量的指针。如果已在固定块中设置指针，则它指向的变量不再是固定的。

下表列出了可在不安全的上下文中对指针执行的运算符和语句：

运算符/语句	用法
*	执行指针间接寻址。
->	通过指针访问结构的成员。
[]	为指针建立索引。
&	获取变量的地址。
++ 和 --	递增和递减指针。
+ 和 -	执行指针算法。
==、!=、<、>、<= 和 >=	比较指针。
stackalloc	在堆栈上分配内存。
fixed 语句	临时固定变量以便找到其地址。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

不安全代码和指针（C# 编程指南）

指针转换（C# 编程指南）

指针表达式（C# 编程指南）

类型（C# 参考）

unsafe（C# 参考）

fixed 语句（C# 参考）

stackalloc（C# 参考）

装箱和取消装箱（C# 编程指南）

XML 文档注释（C# 编程指南）

在 Visual C# 中，你可以通过以下方式为代码创建文档：将特殊注释字段中的 XML 元素包含在源代码中注释引用的代码块的前面，例如：

```
/// <summary>
///  This class performs an important function.
/// </summary>
public class MyClass{}
```

使用 /doc 选项进行编译时，编译器将在源代码中搜索所有 XML 标记，并创建一个 XML 文档文件。若要基于编译器生成的文件创建最终文档，你可以创建一个自定义工具，也可以使用 Sandcastle 等工具。

若要引用 XML 元素（例如，你的函数将处理你要在 XML 文档注释中描述的特定 XML 元素），你可使用标准引用机制（< 和 >）。若要引用代码引用 (**cref**) 元素中的通用标识符，则可使用转义字符（例如，cref="List<T>"）或大括号 (cref="List{T}")。作为特例，编译器会将大括号解析为尖括号以在引用通用标识符时使作者能够更轻松地进行文档注释。

注意
XML 文档注释不是元数据；它们不包括在编译的程序集中，因此无法通过反射对其进行访问。

本节内容

- 建议的文档注释标记
- 处理 XML 文件
- 文档标记分隔符
- 如何：使用 XML 文档功能

相关章节

有关详细信息，请参阅：

- /doc（处理文档注释）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 编程指南

建议的文档注释标记（C# 编程指南）

C# 编译器处理代码中的文档注释，将它们设置为 XML 文件格式，其文件名在 **/doc** 命令行选项中指定。若要创建基于编译器生成的文件的最终文档，您可以创建自定义工具，也可以使用诸如等工具 约定。

在代码构造（如类型和类型成员）上处理标记。

📝 注意
文档注释不能应用于命名空间。

编译器将处理任何为有效 XML 的标记。下列标记提供了用户文档中常用的功能。

标记

<c>	**<para>**	**<see>***
<code>	<param>*	<seealso>*
<example>	<paramref>	<summary>
<exception>*	<permission>*	<typeparam>*
<include>*	<remarks>	<typeparamref>
<list>	<returns>	<value>

（* 表示编译器验证语法。）

如果想要尖括号出现在文档注释文本中，使用 < 和 >。如下面示例所示。

XML

```
/// <summary cref="C &lt; T &gt;">
/// </summary>
```

请参阅

C# 编程指南

/doc (C# Compiler Options)

XML 文档注释（C# 编程指南）

处理 XML 文件（C# 编程指南）

编译器为代码中被标记为生成文档的每一个构造生成一个 ID 字符串。（有关如何标记代码的信息，请参见建议的文档注释标记。）ID 字符串唯一地标识构造。处理 XML 文件的程序可以使用 ID 字符串标识文档应用于的相应 .NET Framework 元数据/反射项。

XML 文件不是代码的分层表示形式，它是一个平面列表，其中的每一个元素都有一个生成的 ID。

编译器在生成 ID 字符串时遵循下列规则：

- 字符串中没有空白。
- ID 字符串的第一部分通过单个字符后跟一个冒号来标识所标识成员的种类。使用下列成员类型：

字符	说明
N	命名空间不可将文档注释添加到命名空间中，但是可以对它们进行 cref 引用（在受支持的位置）。
T	类型：类、接口、结构、枚举、委托
F	字段
P	属性（包括索引程序或其他索引属性）
M	方法（包括一些特殊方法，例如构造函数、运算符等）
E	event
!	错误字符串字符串的其余部分会提供有关此错误的信息。C# 编译器为无法解析的链接生成错误信息。

- 字符串的第二部分是项的完全限定名，从命名空间的根开始。项的名称、其封闭类型和命名空间以句号分隔。如果项的名称本身包含句号，则用哈希符号 ('#') 替换这些句号。假定任何项的名称中都不直接存在哈希符号。例如，String 构造函数的完全限定名是“System.String.#ctor”。
- 对于属性和方法，如果该方法带参数，则将其后的参数列表括在括号中。如果没有参数，则没有括号。多个参数以逗号分隔。每个参数的编码都直接遵循它在 .NET Framework 签名中的编码方法：
    - 基类型。常规类型（ELEMENT_TYPE_CLASS 或 ELEMENT_TYPE_VALUETYPE）表示为类型的完全限定名。

- 内部类型（例如，ELEMENT_TYPE_I4、ELEMENT_TYPE_OBJECT、ELEMENT_TYPE_STRING、ELEMENT_TYPE_TYPEDBYREF。和 ELEMENT_TYPE_VOID）表示为相应完全类型的完全限定名。例如，System.Int32 或 System.TypedReference。
- ELEMENT_TYPE_PTR 表示为“*”，在修改的类型之后。
- ELEMENT_TYPE_BYREF 表示为“@”，在修改的类型之后。
- ELEMENT_TYPE_PINNED 表示为“^”，在修改的类型之后。C# 编译器永远不会生成此结果。
- ELEMENT_TYPE_CMOD_REQ 表示为“|”和修饰符类的完全限定名，在修改的类型之后。C# 编译器永远不会生成此结果。
- ELEMENT_TYPE_CMOD_OPT 表示为“!”和修饰符类的完全限定名，在修改的类型之后。
- ELEMENT_TYPE_SZARRAY 表示为“[]”，在数组的元素类型之后。
- ELEMENT_TYPE_GENERICARRAY 表示为“[?]”，在数组的元素类型之后。C# 编译器永远不会生成此结果。
- ELEMENT*TYPE_ARRAY* 表示为 [_*lowerbound*:*size*,*lowerbound*:*size*]，其中逗号数为秩 - 1，每一维的下限和大小（如果已知）用十进制表示。如果未指定下限及大小，它将完全被省略。如果省略了特定维的下限及大小，则“:”也将被省略。例如，以 1 作为下限并且未指定大小的二维数组是 [1:,1:]。
- ELEMENT*TYPE_FNPTR* 表示为*“=FUNC:_type*(*signature*)”，其中 *type* 是返回类型，*signature* 是方法的参数。如果没有参数，则将省略括号。C# 编译器永远不会生成此结果。

不表示下列签名组件，因为从不使用它们来区分重载方法：

- 调用约定
- 返回类型
- ELEMENT_TYPE_SENTINEL

- 仅对于转换运算符（op_Implicit 和 op_Explicit），方法的返回值才被编码为“~”，后跟返回类型，如上述编码所示。
- 对于泛型类型，类型名称后跟反勾号，再跟一个数字，指示泛型类型参数的个数。例如，

  对于定义为 public class SampleClass<T, U> 的类型，<member name="T:SampleClass`2"> 是其标记。

  对于接受泛型类型作为参数的方法，泛型类型参数被指定为数字前面加上反勾号（如 `0、`1）。表示类型泛型参数的数组表示法（从零开始）的每个数字。

示例

下列示例显示了如何生成类及其成员的 ID 字符串：

```
namespace N
{
    /// <summary>
    /// Enter description here for class X.
    /// ID string generated is "T:N.X".
    /// </summary>
    public unsafe class X
    {
        /// <summary>
        /// Enter description here for the first constructor.
        /// ID string generated is "M:N.X.#ctor".
        /// </summary>
        public X() { }

        /// <summary>
        /// Enter description here for the second constructor.
        /// ID string generated is "M:N.X.#ctor(System.Int32)".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public X(int i) { }

        /// <summary>
        /// Enter description here for field q.
        /// ID string generated is "F:N.X.q".
        /// </summary>
        public string q;

        /// <summary>
        /// Enter description for constant PI.
        /// ID string generated is "F:N.X.PI".
        /// </summary>
        public const double PI = 3.14;

        /// <summary>
        /// Enter description for method f.
        /// ID string generated is "M:N.X.f".
        /// </summary>
        /// <returns>Describe return value.</returns>
        public int f() { return 1; }

        /// <summary>
        /// Enter description for method bb.
        /// ID string generated is "M:N.X.bb(System.String,System.I
        /// </summary>
        /// <param name="s">Describe parameter.</param>
        /// <param name="y">Describe parameter.</param>
        /// <param name="z">Describe parameter.</param>
```

```
        /// <returns>Describe return value.</returns>
        public int bb(string s, ref int y, void* z) { return 1; }

        /// <summary>
        /// Enter description for method gg.
        /// ID string generated is "M:N.X.gg(System.Int16[],System.
        /// </summary>
        /// <param name="array1">Describe parameter.</param>
        /// <param name="array">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int gg(short[] array1, int[,] array) { return 0; }

        /// <summary>
        /// Enter description for operator.
        /// ID string generated is "M:N.X.op_Addition(N.X,N.X)".
        /// </summary>
        /// <param name="x">Describe parameter.</param>
        /// <param name="xx">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static X operator +(X x, X xx) { return x; }

        /// <summary>
        /// Enter description for property.
        /// ID string generated is "P:N.X.prop".
        /// </summary>
        public int prop { get { return 1; } set { } }

        /// <summary>
        /// Enter description for event.
        /// ID string generated is "E:N.X.d".
        /// </summary>
        public event D d;

        /// <summary>
        /// Enter description for property.
        /// ID string generated is "P:N.X.Item(System.String)".
        /// </summary>
        /// <param name="s">Describe parameter.</param>
        /// <returns></returns>
        public int this[string s] { get { return 1; } }

        /// <summary>
        /// Enter description for class Nested.
        /// ID string generated is "T:N.X.Nested".
        /// </summary>
        public class Nested { }

        /// <summary>
        /// Enter description for delegate.
        /// ID string generated is "T:N.X.D".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public delegate void D(int i);
```

```
        /// <summary>
        /// Enter description for operator.
        /// ID string generated is "M:N.X.op_Explicit(N.X)~System.I
        /// </summary>
        /// <param name="x">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static explicit operator int(X x) { return 1; }

    }
}
```

请参阅

C# 编程指南

/doc (C# Compiler Options)

XML 文档注释（C# 编程指南）

文档标记的分隔符（C# 编程指南）

使用 XML doc 注释需要分隔符，分隔符为编译器指示文档注释的起止位置。XML 文档标记可以与以下几种分隔符一起使用：

///

单行分隔符。这是显示在文档示例中并由 Visual C# 项目模板使用的形式。如果有后接分隔符的空白字符，则此字符将不包括在 XML 输出中。

> 注意
>
> Visual Studio IDE 具有一项称为“智能注释编辑”的功能，它自动插入 <summary> 和 </summary> 标记，并且当您在代码编辑器中键入 /// 分隔符后将光标移动到这些标记内。从项目属性页的 选项、文本编辑器、C#、格式设置 可以访问此功能。

/* /

多行分隔符。

以下是使用 /* / 分隔符时要遵循的一些格式设置规则：

- 对于包含 / 分隔符的行，如果该行的其余部分为空白，则不将该行处理为注释。如果 **/** 分隔符后面的第一个字符为空白，则忽略此空白字符并处理该行的剩余部分。否则，将 /** 分隔符后的整行文本处理为注释的一部分。
- 对于包含 / 分隔符的行，如果到 / 分隔符为止仅有空白，则忽略该行。否则，将到 */ 分隔符为止的那行文本处理为注释的一部分，同时必须遵循以下描述的模式匹配规则。
- 对于以 / 分隔符开头的行后面的那些行，编译器会在每个行的开头寻找一个常用模式。该模式可以包含可选的空白和一个星号 **(*)**，后跟多个可选的空白。如果编译器在每行的开头找到不以 **/** 分隔符或 */ 分隔符开头的常见模式，则它将忽略每行的该模式。

下面的示例阐释了这些规则。

- 以下注释中将被处理的唯一部分是以 <summary> 开头的行。这三个标记格式产生相同的注释。

```
/** &lt;summary&gt;text&lt;/summary&gt; */

/**
&lt;summary&gt;text&lt;/summary&gt;
*/

/**
 * &lt;summary&gt;text&lt;/summary&gt;
*/
```

- 编译器在第二行和第三行的开头标识“*”的常见模式。该模式不包括在输出中。

```
/**
 * &lt;summary&gt;
 * text &lt;/summary&gt;*/
```

- 编译器在以下注释中找不到任何常见模式，因为第三行的第二个字符没有星号。因此，第二行和第三行中的所有文本都将被处理为注释的一部分。

```
/**
 * &lt;summary&gt;
   text &lt;/summary&gt;
*/
```

- 编译器之所以在此注释中没找到任何模式有以下两个原因。首先，星号前的空格数并不一致。其次，第五行是以 Tab 开头的，它并不等效于空格。因此，从第二行到第五行的所有文本都将被处理为注释的一部分。

```
/**
  * &lt;summary&gt;
  * text
 *  text2
    *  &lt;/summary&gt;
*/
```

请参阅

C# 编程指南

XML 文档注释（C# 编程指南）

/doc (C# Compiler Options)

XML 文档注释（C# 编程指南）

如何：使用 XML 文档功能（C# 编程指南）

下面的示例提供文档类型的基本概述。

```
// If compiling from the command line, compile with: /doc:YourFileN

/// <summary>
/// Class level summary documentation goes here.</summary>
/// <remarks>
/// Longer comments can be associated with a type or member through
/// the remarks tag.</remarks>
public class TestClass : TestInterface
{
    /// <summary>
    /// Store for the name property.</summary>
    private string _name = null;

    /// <summary>
    /// The class constructor. </summary>
    public TestClass()
    {
        // TODO: Add Constructor Logic here.
    }

    /// <summary>
    /// Name property. </summary>
    /// <value>
    /// A value tag is used to describe the property value.</value>
    public string Name
    {
        get
        {
            if (_name == null)
            {
                throw new System.Exception("Name is null");
            }
            return _name;
        }
    }

    /// <summary>
    /// Description for SomeMethod.</summary>
    /// <param name="s"> Parameter description for s goes here.</pa
    /// <seealso cref="System.String">
    /// You can use the cref attribute on any tag to reference a ty
    /// and the compiler will check that the reference exists. </se
    public void SomeMethod(string s)
    {
    }
```

```
    /// <summary>
    /// Some other method. </summary>
    /// <returns>
    /// Return results are described through the returns tag.</retu
    /// <seealso cref="SomeMethod(string)">
    /// Notice the use of the cref attribute to reference a specifi
    public int SomeOtherMethod()
    {
        return 0;
    }

    public int InterfaceMethod(int n)
    {
        return n * n;
    }

    /// <summary>
    /// The entry point for the application.
    /// </summary>
    /// <param name="args"> A list of command line arguments.</para
    static int Main(System.String[] args)
    {
        // TODO: Add code to start application here.
        return 0;
    }
}

/// <summary>
/// Documentation that describes the interface goes here.
/// </summary>
/// <remarks>
/// Details about the interface go here.
/// </remarks>
interface TestInterface
{
    /// <summary>
    /// Documentation that describes the method goes here.
    /// </summary>
    /// <param name="n">
    /// Parameter n requires an integer argument.
    /// </param>
    /// <returns>
    /// The method returns an integer.
    /// </returns>
    int InterfaceMethod(int n);
}
```

```
//此 .xml 文件生成了与上面的代码示例。<?xml version="1.0"?>
<doc>
    <assembly>
        namexmlsampleAMP_LT/nameAMP_GT
    </assembly>
    <members>
        member name="T:SomeClass"
            <summary>
            Class level summary documentation goes here.</summary>
            <remarks>
            Longer comments can be associated with a type or membe
            通过注释 tagAMP_LT/remarksAMP_GT
        </member>
        member name="F:SomeClass.m_Name"
            <summary>
            名称的 propertyAMP_LT/summaryAMP_GT 存储
        </member>
        member name="M:SomeClass.#ctor"
            summary类 constructor.AMP_LT/summaryAMP_GT
        </member>
        member name="M:SomeClass.SomeMethod(System.String)"
            <summary>
            Description for SomeMethod.</summary>
            param name="s" 中的参数声明转到 hereAMP_LT/paramAMP_GT
            <seealso cref="T:System.String">
            You can use the cref attribute on any tag to reference
            and the compiler will check that the reference exists.
        </member>
        member name="M:SomeClass.SomeOtherMethod"
            <summary>
            Some other method.</summary>
            <returns>
            Return results are described through the returns tag.<
            seealso cref="M:SomeClass.SomeMethod(System.String)"
            通知使用 cref 特性引用特定方法 AMP_LT/seealsoAMP_GT
        </member>
        member name="M:SomeClass.Main(System.String[])"
            <summary>
            The entry point for the application.</summary>
            param name="args" 命令行 argumentsAMP_LT/paramAMP_GT 列
        </member>
        member name="P:SomeClass.Name"
            <summary>
            name 属性 AMP_LT/summaryAMP_GT
            <value>
            值标记用于描述属性 valueAMP_LT/valueAMP_GT
        </member>
    </members>
</doc>
```

编译代码

若要编译该示例，请键入以下命令行:

csc XMLsample.cs /doc:XMLsample.xml

这将创建 XML 文件 XMLsample.xml，使用类型的命令，您可以在浏览器或。

可靠编程

XML 文档以 /// 开头。当您创建新项目时，该向导为您放置一些起始 ///行。对这些注释的处理有一些限制：

- 文档必须是格式良好的 XML。如果 XML 格式不正确，将生成警告，并且文档文件将包含添加的注释遇到错误。
- 开发人员可自由创建自己的标记集。具有建议的标记集 (请参见其他阅读材料部分)。某些建议的标记具有特殊含义：
    - <param> 标记用于描述参数。如果使用，编译器将验证参数是否存在，以及文档中是否描述了所有参数。如果验证失败，编译器会发出警告。
    - **cref** 特性可以附加到任意标记，以提供对代码元素的引用。编译器将验证该代码元素是否存在。如果验证失败，编译器会发出警告。编译器在查找 **cref** 特性中描述的类型时，会考虑所有的 **using** 语句。
    - <summary> 标记由 Visual Studio 内的“Intellisense”使用，用来显示类型或成员的其他相关信息。

注意
XML 文件不提供有关类型和成员的完整信息 (例如，它不包含任何类型信息)。若要获得有关类型或成员的完整信息，文档文件必须与实际类型或成员上的反射一起使用。

请参阅

C# 编程指南

/doc (C# Compiler Options)

XML 文档注释（C# 编程指南）

C# 参考

这部分提供有关 C# 关键字、运算符、编译器错误和警告的参考资料。

本节内容

C# 关键字

提供指向有关 C# 关键字和语法的信息的链接。

C# 运算符

提供指向有关 C# 运算符和语法的信息的链接。

C# 预处理器指令

提供指向有关在 C# 源代码进行嵌入所使用的编译器命令的信息的链接。

C# Compiler Options

包括有关编译器选项以及如何使用这些选项的信息。

C# Compiler Errors

包含一些代码段，演示 C# 编译器错误和警告的原因以更正方法。

C# 语言规范

提供一些链接，供获取 Microsoft Word 格式的最新版本的 C# 语言规范。

相关章节

C# FAQ

在 C# Developer Center 中提供内容不断增加的 C# 常见问题列表。

C# KB articles in the Microsoft Knowledge Base（Microsoft 知识库中的 C# KB 文章）

打开 Microsoft 搜索页可以搜索 MSDN 提供的知识库文章。

Visual C#

提供 Visual C# 文档门户网站。

<paveover>C# Sample Applications

提供 Visual C# 示例的列表和有关如何在本地磁盘上查找这些示例的说明。

使用 Visual C# 开发环境

提供一些链接，这些链接指向描述 IDE 和编辑器的概念性主题及任务主题。

C# 编程指南

包括关于如何使用 C# 编程语言的信息。

C# 关键字

关键字是对编译器具有特殊意义的预定义保留标识符。它们不能在程序中用作标识符，除非它们有一个 @ 前缀。例如，@if 是有效的标识符，但 if 不是，因为 if 是关键字。

本主题中的第一个表列出的关键字在 C# 程序的任何部分都是保留标识符。本主题中的第二个表列出了 C# 中的上下文关键字。上下文关键字仅在受限制的程序上下文中具有特殊含义，并且可在该上下文外部用作标识符。通常，在将新关键字添加到 C# 语言的同时，也会将它们添加为上下文关键字，以便避免破坏用该语言的早期版本编写的程序。

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	Delegate — 委托
do	double	else	enum
事件	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	in（泛型修饰符）	int	interface
internal	为	lock	long
命名空间	new	null	对象
operator	out	out（泛型修饰符）	override
params	private	protected	public
readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc
static	string	struct	switch
this	throw	true	try
typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual
void	volatile	while	

上下文关键字

上下文关键字用于提供代码中的特定含义，但它不是 C# 中的保留字。某些上下文关键字（如 **partial** 和 **where**）在两个或更多个上下文中具有特殊含义。

添加	**Alias — 别名**	**ascending**
async	等待	descending
dynamic	from	get
global	group	into
join	let	orderby
partial（类型）	partial（方法）	remove
select	set	value
var	where（泛型类型约束）	where（查询子句）
yield		

请参阅

C# 参考

C# 编程指南

类型（C# 参考）

C# 类型体系包含下列几种类别：

- 值类型
- 引用类型
- 指针类型

值类型的变量存储数据，而引用类型的变量存储对实际数据的引用。引用类型也称为对象。指针类型仅可用于 unsafe 模式。

通过装箱和取消装箱，可以将值类型转换为引用类型，然后再转换回值类型。除了装箱值类型外，无法将引用类型转换为值类型。

本节还介绍 void 类型。

值类型也可以为 null，这意味着它们能存储其他非值状态。有关更多信息，请参见可以为 null 的类型。

请参阅

C# 参考

C# 编程指南

C# 关键字

类型参考表（C# 参考）

强制转换和类型转换（C# 编程指南）

类型（C# 编程指南）

值类型（C# 参考）

值类型主要由两类组成：

- 结构
- 枚举

结构分为以下几类：

- Numeric（数值）类型
  - 整型
  - 浮点型
  - decimal
- bool
- 用户定义的结构。

值类型的主要功能

基于值类型的变量直接包含值。将一个值类型变量赋给另一个值类型变量时，将复制包含的值。这与引用类型变量的赋值不同，引用类型变量的赋值只复制对对象的引用，而不复制对象本身。

所有的值类型均隐式派生自 System.ValueType。

与引用类型不同，不能从值类型派生出新的类型。但与引用类型相同的是，结构也可以实现接口。

与引用类型不同，值类型无法包含 **null** 值。但是，可以为 null 的类型 功能允许值类型分配给 **null**。

每种值类型均有一个隐式的默认构造函数来初始化该类型的默认值。有关值类型的默认值的信息，请参见默认值表。

简单类型的主要功能

所有的简单类型（C# 语言的组成部分）均为 .NET Framework 系统类型的别名。例如，int 是 System.Int32 的别名。有关完整的别名列表，请参见 内置类型表（C# 参考）。

编译时计算操作数均为简单类型常数的常数表达式。

可使用文字初始化简单类型。例如，“A”是 **char** 类型的文字，而 2001 是 **int** 类型的文字。

初始化值类型

在使用 C# 中的局部变量之前，必须对其进行初始化。例如，可能声明未进行初始化的局部变量，如以下示例所示：

```
int myInt;
```

那么在将其初始化之前，无法使用此变量。可使用下列语句将其初始化：

```
myInt = new int();  // Invoke default constructor for int type.
```

此语句是下列语句的等效语句：

```
myInt = 0;         // Assign an initial value, 0 in this example.
```

当然，可以用同一个语句进行声明和初始化，如下面示例所示：

```
int myInt = new int();
```

- 或 -

```
int myInt = 0;
```

使用 new 运算符时，将调用特定类型的默认构造函数并对变量赋以默认值。在上例中，默认构造函数将值 0 赋给了 myInt。有关通过调用默认构造函数所赋的值的更多信息，请参见默认值表。

对于用户定义的类型，使用 new 来调用默认构造函数。例如，下列语句调用了 Point 结构的默认构造函数：

```
Point p = new Point(); // Invoke default constructor for the struct
```

此调用后，该结构被认为已被明确赋值；也就是说该结构的所有成员均已初始化为各自的默认值。

有关 new 运算符的更多信息，请参见 new。

有关格式化数字类型输出的信息，请参见格式化数值结果表。

请参阅

C# 参考

C# 编程指南

C# 关键字

类型（C# 参考）

类型参考表（C# 参考）

引用类型（C# 参考）

bool（C# 参考）

bool 关键字是 System.Boolean 的别名。它用于声明变量来存储布尔值 true 和 false。

注意
如果需要一个也可以有 **null** 值的布尔型变量，请使用 bool?。有关更多信息，请参见 可以为 null 的类型（C# 编程指南）。

文本

可将布尔值赋给 **bool** 变量。也可以将计算结果为 **bool** 类型的表达式赋给 **bool** 变量。

```
public class BoolTest
{
    static void Main()
    {
        bool b = true;

        // WriteLine automatically converts the value of b to text.
        Console.WriteLine(b);

        int days = DateTime.Now.DayOfYear;

        // Assign the result of a boolean expression to b.
        b = (days % 2 == 0);

        // Branch depending on whether b is true or false.
        if (b)
        {
            Console.WriteLine("days is an even number");
        }
        else
        {
            Console.WriteLine("days is an odd number");
        }
    }
}
/* Output:
  True
  days is an <even/odd> number
*/
```

bool 变量的默认值为 **false**。 **bool?** 变量的默认值为 **null**。

转换

在 C++ 中，**bool** 类型的值可转换为 **int** 类型的值；也就是说，**false** 等效于零值，而 **true** 等效于非零值。在 C# 中，不存在 **bool** 类型与其他类型之间的相互转换。例如，下面的 **if** 语句在 C# 中无效：

```
int x = 123;

// if (x)   // Error: "Cannot implicitly convert type 'int' to 'boo
{
    Console.Write("The value of x is nonzero.");
}
```

若要测试 **int** 类型的变量，必须将该变量与一个值（例如零）进行显式比较，如下所示：

```
if (x != 0)   // The C# way
{
    Console.Write("The value of x is nonzero.");
}
```

在此例中，您从键盘输入一个字符，然后程序检查输入的字符是否是一个字母。如果字符是一个字母，则程序检查它是大写还是小写。这些检查是使用 IsLetter 和 IsLower（两者均返回 **bool** 类型）来执行的：

```
public class BoolKeyTest
{
    static void Main()
    {
        Console.Write("Enter a character: ");
        char c = (char)Console.Read();
        if (Char.IsLetter(c))
        {
            if (Char.IsLower(c))
            {
                Console.WriteLine("The character is lowercase.");
            }
            else
            {
                Console.WriteLine("The character is uppercase.");
            }
        }
        else
        {
            Console.WriteLine("Not an alphabetic character.");
        }
    }
}
/* Sample Output:
    Enter a character: X
    The character is uppercase.

    Enter a character: x
    The character is lowercase.

    Enter a character: 2
    The character is not an alphabetic character.
 */
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

byte（C# 参考）

byte 关键字代表一种整型，该类型按下表所示存储值：

类型	范围	大小	.NET Framework 类型
byte	0 到 255	无符号 8 位整数	System.Byte

文本

可如下例所示声明并初始化 **byte** 类型的变量：

```
byte myByte = 255;
```

在以上声明中，整数 255 从 int 隐式转换为 **byte**。如果整数超出了 **byte** 的范围，将产生编译错误。

转换

存在从 **byte** 到 short、ushort、int、uint、long、ulong、float、double 或 decimal 的预定义隐式转换。

不能将更大存储大小的非文本数值类型隐式转换为 **byte**。有关整型的存储大小的更多信息，请参见 整型表（C# 参考）。例如，请看以下两个 **byte** 变量 x 和 y：

```
byte x = 10, y = 20;
```

以下赋值语句将产生一个编译错误，原因是赋值运算符右侧的算术表达式在默认情况下的计算结果为 **int** 类型。

```
// Error: conversion from int to byte:
byte z = x + y;
```

若要解决此问题，请使用强制转换：

```
// OK: explicit conversion:
byte z = (byte)(x + y);
```

但是，在目标变量具有相同或更大的存储大小时，使用下列语句是可能的：

```
int x = 10, y = 20;
int m = x + y;
long n = x + y;
```

同样，不存在从浮点型到 **byte** 类型的隐式转换。例如，除非使用显式强制转换，否则以下语句将生成一个编译器错误：

```
// Error: no implicit conversion from double:
byte x = 3.0;
// OK: explicit conversion:
byte y = (byte)3.0;
```

调用重载方法时，必须使用显式转换。以下面使用 **byte** 和 int 参数的重载方法为例：

```
public static void SampleMethod(int i) {}
public static void SampleMethod(byte b) {}
```

使用 **byte** 强制转换可保证调用正确的类型，例如：

```
// Calling the method with the int parameter:
SampleMethod(5);
// Calling the method with the byte parameter:
SampleMethod((byte)5);
```

有关兼用浮点型和整型的算术表达式的信息，请参见 float 和 double。

有关隐式数值转换规则的更多信息，请参见 隐式数值转换表（C# 参考）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

Byte

C# 参考

C# 编程指南

C# 关键字

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

char（C# 参考）

char 关键字用于声明 .NET framework 使用 Unicode 字符表示 System.Char 结构的实例。 **Char** 对象的值是 16 位数字 (序号值。)

Unicode 字符在世界上表示大多数书面语言。

类型	范围	大小	.NET Framework 类型
char	U+0000 到 U+FFFF	16 位 Unicode 字符	System.Char

文本

char 类型的常数可以写成字符、十六进制换码序列或 Unicode 表示形式。您也可以显式转换整数字符代码。在下面的示例中，四个 **char** 变量使用同一字符 X 初始化：

```
char[] chars = new char[4];

chars[0] = 'X';        // Character literal
chars[1] = '\x0058';   // Hexadecimal
chars[2] = (char)88;   // Cast from integral type
chars[3] = '\u0058';   // Unicode

foreach (char c in chars)
{
    Console.Write(c + " ");
}
// Output: X X X X
```

转换

char 可以隐式转换为 ushort、int、uint、long、ulong、float、double 或 decimal。但是，不存在从其他类型到 **char** 类型的隐式转换。

System.Char 类型提供几个处理 **char** 值的静态方法。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

Char

C# 参考

C# 编程指南

C# 关键字

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

可以为 null 的类型（C# 编程指南）

字符串（C# 编程指南）

decimal（C# 参考）

decimal 关键字指示 128 位数据类型。与浮点型相比，**decimal** 类型具有更高的精度和更小的范围，这使它适合于财务和货币计算。 **decimal** 类型的大致范围和精度如下表所示。

类型	大致范围	精度	.NET Framework 类型
decimal	(-7.9 x 1028 - 7.9 x 1028) / (100 - 28)	28-29 个有效位	System.Decimal

文本

如果希望实数被视为 **decimal** 类型，请使用后缀 m 或 M，例如：

```
decimal myMoney = 300.5m;
```

如果没有后缀 m，则数字将被视为 double 类型并会生成编译器错误。

转换

整型将被隐式转换为 **decimal** 类型，其计算结果为 **decimal**。因此，你可以使用整数文本初始化十进制变量而不使用后缀，如下所示：

```
decimal myMoney = 300;
```

在浮点型和 **decimal** 类型之间不存在隐式转换；因此，必须使用强制转换以在这两个类型之间转换。例如：

```
        decimal myMoney = 99.9m;
double x = (double)myMoney;
myMoney = (decimal)x;
```

你还可以在同一表达式中混合使用 **decimal** 和数值整型。但是，不进行强制转换就混合使用 **decimal** 和浮点型将导致编译错误。

有关隐式数值转换的更多信息，请参见隐式数值转换表（C# 参考）。

有关显式数值转换的更多信息，请参见显式数值转换表（C# 参考）。

设置十进制输出的格式

你可以通过使用 **String.Format** 方法或 Console.Write 方法（其调用 **String.Format()**）来设置结果的格式。货币格式是使用标准货币格式字符串“C”或“c”指定的，如本文后面的第二个示例所示。有关 **String.Format** 方法的更多信息，请参见 String.Format。

下面的示例尝试添加 double 和 **decimal** 变量，这会导致编译器错误。

```
double dub = 9;
// The following line causes an error that reads "Operator '+' cann
// operands of type 'double' and 'decimal'"
Console.WriteLine(dec + dub);

// You can fix the error by using explicit casting of either operan
Console.WriteLine(dec + (decimal)dub);
Console.WriteLine((double)dec + dub);
```

其结果为以下错误：

Operator '+' cannot be applied to operands of type 'double' and 'decimal'

在此示例中，同一个表达式中混合使用了 **decimal** 和 int。计算结果为 **decimal** 类型。

```
public class TestDecimal
{
    static void Main()
    {
        decimal d = 9.1m;
        int y = 3;
        Console.WriteLine(d + y);   // Result converted to decimal
    }
}
// Output: 12.1
```

在此示例中，通过使用货币格式字符串来设置输出的格式。请注意，x 被舍入，因为其小数位数超出了 $0.99。表示最大精确位数的变量 y 严格按照正确的格式显示。

```
public class TestDecimalFormat
{
    static void Main()
    {
        decimal x = 0.999m;
        decimal y = 9999999999999999999999999999m;
        Console.WriteLine("My amount = {0:C}", x);
        Console.WriteLine("Your amount = {0:C}", y);
    }
}
/* Output:
    My amount = $1.00
    Your amount = $9,999,999,999,999,999,999,999,999,999.00
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

Decimal

C# 参考

C# 编程指南

C# 关键字

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

标准数字格式字符串

double（C# 参考）

double 关键字表示存储 64 位浮点值的简单类型。下表显示了 **double** 类型的精度和大致范围。

类型	大致范围	精度	.NET Framework 类型
double	±5.0 × 10−324 到 ±1.7 × 10308	15 到 16 位	System.Double

文本

默认情况下，赋值运算符右侧的实数被视为 **double**。但是，如果希望整数被视为 **double**，请使用后缀 d 或 D，例如：

```
double x = 3D;
```

转换

可在一个表达式中兼用数值整型和浮点型。在此情况下，整型将转换为浮点型。根据以下规则计算表达式：

- 如果其中一个浮点类型为 **double**，则表达式的计算结果为 **double** 或 bool（在关系表达式或布尔表达式中）。
- 如果表达式中不存在 **double** 类型，则表达式的计算结果为 float 或 bool（在关系表达式或布尔表达式中）。

浮点表达式可以包含下列值集：

- 正零和负零。
- 正无穷和负无穷。
- 非数字值 (NaN)。
- 有限的非零值集。

有关这些值的更多信息，请参见 IEEE 网站上的“IEEE Standard for Binary Floating-Point Arithmetic”（二进制浮点算法的 IEEE 标准）。

在下面的示例中，一个 int、一个 short、一个 float 和一个 **double** 相加，计算结果为 **double** 类型。

```
// Mixing types in expressions
class MixedTypes
{
    static void Main()
    {
        int x = 3;
        float y = 4.5f;
        short z = 5;
        double w = 1.7E+3;
        // Result of the 2nd argument is a double:
        Console.WriteLine("The sum is {0}", x + y + z + w);
    }
}
// Output: The sum is 1712.5
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

默认值表（C# 参考）

内置类型表（C# 参考）

浮点型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

enum（C# 参考）

enum 关键字用于声明枚举，即一种由一组称为枚举数列表的命名常量组成的独特类型。

通常情况下，最好是在命名空间内直接定义枚举，以便该命名空间中的所有类都能够同样方便地访问它。但是，还可以将枚举嵌套在类或结构中。

默认情况下，第一个枚举数的值为 0，后面每个枚举数的值依次递增 1。例如，下面的枚举，Sat 是 0，Sun 是 1，Mon 是 2 等.

```
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

如下面的示例所示枚举数可用初始值来重写默认值。

```
enum Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
```

在此枚举中，强制元素序列从 1 而不是 0 开始。但是，一般建议包括值为 0 的常量。有关更多信息，请参见 枚举类型（C# 编程指南）。

每种枚举类型都有基础类型，该类型可以是除 char 以外的任何整型。枚举元素的默认基础类型为 int。要声明另一整型枚举（如 byte），请在标识符之后紧跟类型，然后再使用冒号，如下面的示例所示。

```
enum Days : byte {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
```

准许使用的枚举类型有 **byte**、sbyte、short、ushort、int、uint、long 或 ulong。

可以给 Days 类型的变量赋以基础类型范围内的任何值，所赋的值不限于已命名的常数。

enum E 的默认值为表达式 (E)0 生成的值。

注意
枚举数的名称中不能包含空白。

基础类型指定为每个枚举数分配的存储大小。但是，从 **enum** 类型到整型的转换需要用显式类型转换来完成。例如，下面的语句使用强制转换（从 **enum** 转换为 **int**）将枚举数 Sun 赋值给一个 int 类型的变量。

```
int x = (int)Days.Sun;
```

将 System.FlagsAttribute 应用于某个枚举时，如果该枚举包含一些使用按位 **OR** 运算组合的元素，该特性在用于某些工具时会影响 **enum** 的行为。当使用诸如 Console 类方法、表达式计算器这样的工具时，可以注意到这些变化。（请参见第三个示例。）

可靠编程

与任何常量一样，对枚举中各个值的所有引用在编译时均将转换为数值文本。这可形成潜在的版本控制问题，如常量（C# 编程指南）中所述。

给新版本的枚举赋其他值，或者更改新版本中枚举成员的值，可导致相关源代码出现问题。通常在 switch 语句中使用枚举值。如果 **enum** 类型中添加了其他元素，则 switch 语句的默认节可能被意外选定。

如果其他开发人员使用您的代码，则需要提供相关说明，告诉开发人员如果任何 **enum** 类型中添加了新元素，他们的代码应该如何响应。

在下面的示例中，将声明枚举 Days。两个枚举数被显式转换为整数并赋给整型变量。

```
public class EnumTest
{
    enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

    static void Main()
    {
        int x = (int)Days.Sun;
        int y = (int)Days.Fri;
        Console.WriteLine("Sun = {0}", x);
        Console.WriteLine("Fri = {0}", y);
    }
}
/* Output:
   Sun = 0
   Fri = 5
*/
```

在下面例中，使用了基类选项来声明 **long** 类型的 **enum** 成员。请注意，即使枚举的基础类型是 **long**，也仍然必须使用强制转换将枚举成员显式转换为 **long** 类型。

```
public class EnumTest2
{
    enum Range : long { Max = 2147483648L, Min = 255L };
    static void Main()
    {
        long x = (long)Range.Max;
        long y = (long)Range.Min;
        Console.WriteLine("Max = {0}", x);
        Console.WriteLine("Min = {0}", y);
    }
}
/* Output:
   Max = 2147483648
   Min = 255
*/
```

下面的代码示例阐释 **enum** 声明上的 System.FlagsAttribute 特性的使用和效果。

```
// Add the attribute Flags or FlagsAttribute.
[Flags]
public enum CarOptions
{
    // The flag for SunRoof is 0001.
    SunRoof = 0x01,
    // The flag for Spoiler is 0010.
    Spoiler = 0x02,
    // The flag for FogLights is 0100.
    FogLights = 0x04,
    // The flag for TintedWindows is 1000.
    TintedWindows = 0x08,
}

class FlagTest
{
    static void Main()
    {
        // The bitwise OR of 0001 and 0100 is 0101.
        CarOptions options = CarOptions.SunRoof | CarOptions.FogLi

        // Because the Flags attribute is specified, Console.WriteI
        // the name of each enum element that corresponds to a flag
        // the value 1 in variable options.
        Console.WriteLine(options);
        // The integer value of 0101 is 5.
        Console.WriteLine((int)options);
    }
}
/* Output:
   SunRoof, FogLights
   5
*/
```

注释

如果您移除 **Flags**，则示例将显示以下值：

5

5

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

枚举类型（C# 编程指南）

C# 关键字

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

float（C# 参考）

float 关键字表示存储 32 位浮点值的简单类型。下表显示了 **float** 类型的精度和大致范围。

类型	大致范围	精度	.NET Framework 类型
float	-3.4 × 1038 到 +3.4 × 1038	7 位	System.Single

文本

默认情况下，赋值运算符右侧的实数被视为 double。因此，应使用后缀 f 或 F 初始化浮点型变量，如以下示例中所示：

```
float x = 3.5F;
```

如果在以上声明中不使用后缀，则会因为您尝试将一个 double 值存储到 **float** 变量中而发生编译错误。

转换

可在一个表达式中兼用数值整型和浮点型。在此情况下，整型将转换为浮点型。根据以下规则计算表达式：

- 如果其中一个浮点型为 double，则表达式的计算结果为 double 或 bool（在关系表达式或布尔表达式中）。
- 如果表达式中不存在 double 类型，则表达式的计算结果为 **float** 或 bool（在关系表达式或布尔表达式中）。

浮点表达式可以包含下列值集：

- 正零和负零
- 正无穷和负无穷
- 非数字值 (NaN)
- 有限的非零值集

有关这些值的更多信息，请参见 IEEE 网站上的“IEEE Standard for Binary Floating-Point Arithmetic”（二进制浮点算法的 IEEE 标准）。

在下面的示例中，包含 int、short 和 **float** 类型的数学表达式得到一个 **float** 结果。（请记住 **float** 是 System.Single 类型的别名。）请注意，表达式中没有 double。

```
class FloatTest
{
    static void Main()
    {
        int x = 3;
        float y = 4.5f;
        short z = 5;
        var result = x * y / z;
        Console.WriteLine("The result is {0}", result);
        Type type = result.GetType();
        Console.WriteLine("result is of type {0}", type.ToString()
    }
}
/* Output:
  The result is 2.7
  result is of type System.Single //'float' is alias for 'Single'
 */
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

Single

C# 参考

C# 编程指南

强制转换和类型转换（C# 编程指南）

C# 关键字

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

int（C# 参考）

int 关键字表示一种整型，该类型根据下表显示的大小和范围存储值。

类型	范围	大小	.NET Framework 类型	默认值
int	-2,147,483,648 到 2,147,483,647	有符号 32 位整数	System.Int32	0

文本

可以声明并初始化 **int** 类型的变量，例如：

```
int i = 123;
```

如果整数没有后缀，则其类型为以下类型中可表示其值的第一个类型：**int**、uint、long、ulong。在此例中为 **int** 类型。

转换

存在从 **int** 到 long、float、double 或 decimal 的预定义隐式转换。例如：

```
// '123' is an int, so an implicit conversion takes place here:
float f = 123;
```

存在从 sbyte、byte、short、ushort 或 char 到 **int** 的预定义隐式转换。例如，如果不进行强制转换，下面的赋值语句将产生编译错误：

```
long aLong = 22;
int i1 = aLong;       // Error: no implicit conversion from long.
int i2 = (int)aLong;  // OK: explicit conversion.
```

还请注意，不存在从浮点型到 **int** 类型的隐式转换。例如，除非使用显式强制转换，否则以下语句将生成一个编译器错误：

```
        int x = 3.0;         // Error: no implicit conversion from
int y = (int)3.0;    // OK: explicit conversion.
```

有关兼用浮点型和整型的算术表达式的更多信息，请参见 float 和 double。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

Int32

C# 参考

C# 编程指南

C# 关键字

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

long（C# 参考）

long 关键字表示一种整型，该类型根据下表显示的大小和范围存储值。

类型	范围	大小	.NET Framework 类型
long	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807	有符号 64 位整数	System.Int64

文本

可如下例所示声明并初始化 **long** 类型的变量：

```
long long1 = 4294967296;
```

如果整数没有后缀，则其类型为以下类型中可表示其值的第一个类型：int、uint、**long**、ulong。在上例中，它是 **long** 类型，因为它超出了 uint 的范围（有关整型的存储大小，请参见 整型表（C# 参考））。

还可以像下面这样，在 **long** 类型中使用后缀 L：

```
long long2 = 4294967296L;
```

当使用后缀 L 时，将根据整数的大小确定它的类型为 **long** 还是 ulong。在此例中，它是 **long**，因为它小于 ulong 的范围的下限。

此后缀常用于调用重载方法。以下面使用 **long** 和 int 参数的重载方法为例：

```
public static void SampleMethod(int i) {}
public static void SampleMethod(long l) {}
```

使用后缀 L 可保证调用正确的类型，例如：

```
SampleMethod(5);    // Calling the method with the int parameter
SampleMethod(5L);   // Calling the method with the long parameter
```

可在同一个表达式中同时使用 **long** 类型和其他数值整型，这时表达式的计算结果为 **long**（在关系表达式或布尔表达式中为 bool）类型。例如，下列表达式计算为 **long**：

```
898L + 88
```

> 注意
>
> 也可用小写字母“l”作后缀。但是，因为字母“l”容易与数字“1”混淆，会生成编译器警告。为清楚起见，请使用“L”。

有关兼用浮点型和整型的算术表达式的信息，请参见 float 和 double。

转换

存在从 **long** 到 float、double 或 decimal 的预定义隐式转换。其他情况下必须使用显式转换。例如，不使用显式类型转换时，下列语句将产生编译错误：

```
int x = 8L;        // Error: no implicit conversion from long to ir
int x = (int)8L;   // OK: explicit conversion to int
```

存在从 sbyte、byte、short、ushort、int、uint 或 char 到 **long** 的预定义隐式转换。

还请注意，不存在从浮点型到 **long** 类型的隐式转换。例如，除非使用显式强制转换，否则以下语句将生成一个编译器错误：

```

        long x = 3.0;          // Error: no implicit conversion fror
long y = (long)3.0;   // OK: explicit conversion
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

Int64

C# 参考

C# 编程指南

C# 关键字

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

sbyte（C# 参考）

sbyte 关键字表示一种整型，该类型根据下表显示的大小和范围存储值。

类型	范围	大小	.NET Framework 类型
sbyte	-128 到 127	有符号 8 位整数	System.SByte

文本

可使用下述方法声明并初始化 **sbyte** 类型的变量：

```
sbyte sByte1 = 127;
```

在以上声明中，整数 127 从 int 隐式转换为 **sbyte**。如果整数超出了 **sbyte** 的范围，将产生编译错误。

调用重载方法时必须使用强制转换。以下面使用 **sbyte** 和 int 参数的重载方法为例：

```
public static void SampleMethod(int i) {}
public static void SampleMethod(sbyte b) {}
```

使用 **sbyte** 强制转换可保证调用正确的类型，例如：

```
// Calling the method with the int parameter:
SampleMethod(5);
// Calling the method with the sbyte parameter:
SampleMethod((sbyte)5);
```

转换

存在从 **sbyte** 到 short、int、long、float、double 或 decimal 的预定义隐式转换。

不能将存储大小更大的非文本数值类型隐式转换为 **sbyte** 类型（有关整型的存储大小的信息，请参见 整型表（C# 参考））。例如，请看以下两个 **sbyte** 变量 x 和 y：

```
sbyte x = 10, y = 20;
```

以下赋值语句将产生一个编译错误，原因是赋值运算符右侧的算术表达式在默认情况下的计算结果为 int。

```
sbyte z = x + y;   // Error: conversion from int to sbyte
```

若要更正此问题，请对该表达式执行强制转换，如下例所示：

```
sbyte z = (sbyte)(x + y);   // OK: explicit conversion
```

但是，在目标变量具有相同或更大的存储大小时，使用下列语句是可能的：

```
        sbyte x = 10, y = 20;
int m = x + y;
long n = x + y;
```

还请注意，不存在从浮点型到 **sbyte** 类型的隐式转换。例如，除非使用显式强制转换，否则以下语句将生成一个编译器错误：

```
        sbyte x = 3.0;         // Error: no implicit conversion fro
sbyte y = (sbyte)3.0;  // OK: explicit conversion
```

有关兼用浮点型和整型的算术表达式的信息，请参见 float 和 double。

有关隐式数值转换规则的更多信息，请参见隐式数值转换表（C# 参考）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

SByte

C# 参考

C# 编程指南

C# 关键字

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

short（C# 参考）

short 关键字表示一种整数数据类型，该类型根据下表显示的大小和范围存储值。

类型	范围	大小	.NET Framework 类型
short	-32,768 到 32,767	有符号 16 位整数	System.Int16

文本

可如下例所示声明并初始化 **short** 类型的变量：

```
short x = 32767;
```

在以上声明中，整数 32767 从 int 隐式转换为 **short**。 如果整数的长度超过了 **short** 存储位置的大小，则将产生编译错误。

调用重载方法时必须使用强制转换。 以下面使用 **short** 和 int 参数的重载方法为例：

```
public static void SampleMethod(int i) {}
public static void SampleMethod(short s) {}
```

使用 **short** 强制转换可保证调用正确的类型，例如：

```
SampleMethod(5);         // Calling the method with the int paramet
SampleMethod((short)5);  // Calling the method with the short parar
```

转换

存在从 **short** 到 int、long、float、double 或 decimal 的预定义隐式转换。

不能将存储大小更大的非文本数值类型隐式转换为 **short** 类型（有关整型的存储大小的信息，请参见 整型表（C# 参考））。 例如，请看以下两个 **short** 变量 x 和 y：

```
short x = 5, y = 12;
```

以下赋值语句将产生一个编译错误，原因是赋值运算符右侧的算术表达式在默认情况下的计算结果为 int 类型。

short z = x + y; // Error: no conversion from int to short

若要解决此问题，请使用强制转换：

short z = (**short**)(x + y); // OK: explicit conversion

但是，在目标变量具有相同或更大的存储大小时，使用下列语句是可能的：

```
int m = x + y;
long n = x + y;
```

不存在从浮点型到 **short** 类型的隐式转换。 例如，除非使用显式强制转换，否则以下语句将生成一个编译器错误：

```
        short x = 3.0;          // Error: no implicit conversion fr
short y = (short)3.0;   // OK: explicit conversion
```

有关兼用浮点型和整型的算术表达式的信息，请参见 float 和 double。

有关隐式数值转换规则的更多信息，请参见 隐式数值转换表（C# 参考）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

Int16

C# 参考

C# 编程指南

C# 关键字

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

struct（C# 参考）

struct 类型是一种值类型，通常用来封装小型相关变量组，例如，矩形的坐标或库存商品的特征。下面的示例显示了一个简单的结构声明：

```
public struct Book
{
    public decimal price;
    public string title;
    public string author;
}
```

备注

结构还可以包含构造函数、常量、字段、方法、属性、索引器、运算符、事件和嵌套类型，但如果同时需要上述几种成员，则应当考虑改为使用类作为类型。

有关示例，请参阅使用结构（C# 编程指南）。

结构可以实现接口，但它们无法继承另一个结构。因此，结构成员无法声明为 **protected**。

有关详细信息，请参阅结构（C# 编程指南）。

示例

有关示例和详细信息，请参阅 使用结构（C# 编程指南）。

C# 语言规范

有关示例，请参阅使用结构（C# 编程指南）。

请参阅

C# 参考

C# 编程指南

C# 关键字

默认值表（C# 参考）

内置类型表（C# 参考）

类型（C# 参考）

值类型（C# 参考）

class（C# 参考）

接口（C# 参考）

类和结构（C# 编程指南）

uint（C# 参考）

uint 关键字表示一种整型，该类型根据下表显示的大小和范围存储值。

类型	范围	大小	.NET Framework 类型
uint	0 到 4,294,967,295	无符号 32 位整数	System.UInt32

注意 **uint** 类型与 CLS 不兼容。应尽可能使用 **int**。

文本

可如下例所示声明并初始化 **uint** 类型的变量：

```
uint myUint = 4294967290;
```

如果整数没有后缀，则其类型为以下类型中可表示其值的第一个类型：int、**uint**、long、ulong。在此例中，它是 **uint**：

```
uint uInt1 = 123;
```

还可以像下面这样使用后缀 u 或 U：

```
uint uInt2 = 123U;
```

当使用后缀 U 或 u 时，将根据文本的数值来确定文本的类型是 **uint** 还是 **ulong**。例如：

```
Console.WriteLine(44U.GetType());
Console.WriteLine(323442434344U.GetType());
```

此代码先后显示 System.UInt32 和 System.UInt64（它们分别是 **uint** 和 **ulong** 的基础类型），因为第二个文本太大，无法用 **uint** 类型来存储。

转换

存在从 **uint** 到 long、ulong、float、double 或 decimal 的预定义隐式转换。例如：

```
float myFloat = 4294967290;   // OK: implicit conversion to float
```

存在从 byte、ushort 或 char 到 **uint** 的预定义隐式转换。否则必须使用显式转换。例如，如果不进行强制转换，下面的赋值语句将产生编译错误：

```
long aLong = 22;
// Error -- no implicit conversion from long:
uint uInt1 = aLong;
// OK -- explicit conversion:
uint uInt2 = (uint)aLong;
```

还请注意，不存在从浮点型到 **uint** 类型的隐式转换。例如，除非使用显式强制转换，否则以下语句将生成一个编译器错误：

```
// Error -- no implicit conversion from double:
uint x = 3.0;
// OK -- explicit conversion:
uint y = (uint)3.0;
```

有关兼用浮点型和整型的算术表达式的信息，请参见 float 和 double。

有关隐式数值转换规则的更多信息，请参见隐式数值转换表（C# 参考）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

UInt32

C# 参考

C# 编程指南

C# 关键字

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

ulong（C# 参考）

ulong 关键字表示一种整型，该类型根据下表显示的大小和范围存储值。

类型	范围	大小	.NET Framework 类型
ulong	0 到 18,446,744,073,709,551,615	无符号 64 位整数	System.UInt64

文本

可如下例所示声明并初始化 **ulong** 类型的变量：

```
ulong uLong = 9223372036854775808;
```

如果整数没有后缀，则其类型为以下类型中可表示其值的第一个类型：int、uint、long、**ulong**。在上面的示例中，它是 **ulong** 类型。

还可根据以下规则使用后缀指定文字类型：

- 如果使用 L 或 l，那么根据整数的大小，可以判断出其类型为 long 还是 **ulong**。

注意
注意也可用小写字母“l”作后缀。但是，因为字母“l”容易与数字“1”混淆，会生成编译器警告。为清楚起见，请使用“L”。

- 如果使用 U 或 u，那么根据整数的大小，可以判断出其类型为 uint 还是 **ulong**。
- 如果使用 UL、ul、Ul、uL、LU、lu、Lu 或 lU，则整数的类型为 **ulong**。

  例如，以下三个语句的输出将为系统类型 **UInt64**，此类型对应于别名 **ulong**：

  ```
  Console.WriteLine(9223372036854775808L.GetType());
  Console.WriteLine(123UL.GetType());
  Console.WriteLine((123UL + 456).GetType());
  ```

此后缀常用于调用重载方法。以下面使用 **ulong** 和 int 参数的重载方法为例：

```
public static void SampleMethod(int i) {}
public static void SampleMethod(ulong l) {}
```

在 **ulong** 参数后加上后缀可保证调用正确的类型，例如：

```
SampleMethod(5);    // Calling the method with the int parameter
SampleMethod(5UL);  // Calling the method with the ulong parameter
```

转换

存在从 **ulong** 到 float、double 或 decimal 的预定义隐式转换。

不存在从 **ulong** 到任何整型的隐式转换。例如，不使用显式类型转换时，下列语句将产生编译错误：

```
long long1 = 8UL;   // Error: no implicit conversion from ulong
```

存在从 byte、ushort、uint 或 char 到 **ulong** 的预定义隐式转换。

同样，不存在从浮点型到 **ulong** 类型的隐式转换。例如，除非使用显式强制转换，否则以下语句将生成一个编译器错误：

```
// Error -- no implicit conversion from double:
ulong x = 3.0;
// OK -- explicit conversion:
ulong y = (ulong)3.0;
```

有关兼用浮点型和整型的算术表达式的信息，请参见 float 和 double。

有关隐式数值转换规则的更多信息，请参见 隐式数值转换表（C# 参考）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

UInt64

C# 参考

C# 编程指南

C# 关键字

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

ushort（C# 参考）

ushort 关键字表示整数数据类型，该类型根据下表显示的大小和范围存储值。

类型	范围	大小	.NET Framework 类型
ushort	0 到 65,535	无符号 16 位整数	System.UInt16

文本

可如下例所示声明并初始化 **ushort** 类型的变量：

```
ushort myShort = 65535;
```

在以上声明中，整数 65535 从 int 隐式转换为 **ushort**。如果整数超出了 **ushort** 的范围，将产生编译错误。

调用重载方法时，必须使用强制转换。以下面使用 **ushort** 和 int 参数的重载方法为例：

```
public static void SampleMethod(int i) {}
public static void SampleMethod(ushort s) {}
```

使用 **ushort** 强制转换可保证调用正确的类型，例如：

```
// Calls the method with the int parameter:
SampleMethod(5);
// Calls the method with the ushort parameter:
SampleMethod((ushort)5);
```

转换

存在从 **ushort** 到 int、uint、long、ulong、float、double 或 decimal 的预定义隐式转换。

存在从 byte 或 char 到 **ushort** 的预定义隐式转换。其他情况下必须使用显式转换。例如，请看以下两个 **ushort** 变量 x 和 y：

```
ushort x = 5, y = 12;
```

以下赋值语句将产生一个编译错误，原因是赋值运算符右侧的算术表达式在默认情况下的计算结果为 **int**。

```
ushort z = x + y;   // Error: conversion from int to ushort
```

若要解决此问题，请使用强制转换：

```
ushort z = (ushort)(x + y);   // OK: explicit conversion
```

但是，在目标变量具有相同或更大的存储大小时，使用下列语句是可能的：

```
int m = x + y;
long n = x + y;
```

还请注意，不存在从浮点型到 **ushort** 类型的隐式转换。例如，除非使用显式强制转换，否则以下语句将生成一个编译器错误：

```
// Error -- no implicit conversion from double:
ushort x = 3.0;
// OK -- explicit conversion:
ushort y = (ushort)3.0;
```

有关兼用浮点型和整型的算术表达式的信息，请参见 float 和 double。

有关隐式数值转换规则的更多信息，请参见隐式数值转换表（C# 参考）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

UInt16

C# 参考

C# 编程指南

C# 关键字

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

显式数值转换表（C# 参考）

引用类型（C# 参考）

C# 中有两种类型：引用类型和值类型。引用类型的变量存储对其数据（对象）的引用，而值类型的变量直接包含其数据。对于引用类型，两种变量可引用同一对象；因此，对一个变量执行的操作会影响另一个变量所引用的对象。对于值类型，每个变量都具有其自己的数据副本，对一个变量执行的操作不会影响另一个变量（ref 和 out 参数变量除外，请参见 ref（C# 参考）和 out 参数修饰符（C# 参考））。

下列关键字用于声明引用类型：

- class
- interface
- delegate

C# 也提供了下列内置引用类型：

- dynamic
- object
- string

请参阅

C# 参考

C# 编程指南

C# 关键字

类型（C# 参考）

值类型（C# 参考）

class（C# 参考）

类是使用关键字 **class** 声明的，如下面的示例所示：

```
      class TestClass
{
    // Methods, properties, fields, events, delegates
    // and nested classes go here.
}
```

备注

唯一继承在 c# 中允许的。也就是说，类只能从一个基类继承实现。但是，一个类可以实现一个以上的接口。下表给出了类继承和接口实现的一些示例：

Inheritance	示例
无	`class ClassA { }`
Single	`class DerivedClass: BaseClass { }`
无，实现两个接口	`class ImplClass: IFace1, IFace2 { }`
单一，实现一个接口	`class ImplDerivedClass: BaseClass, IFace1 { }`

在其他选件类中可以直接在命名空间内声明的选件类，未嵌套，可以是 公开 或 内部。默认情况下选件类是 **internal**。

类成员，包括嵌套选件类，可以是 公开、**protected internal**、保护、内部或 专用。默认情况下成员是 专用。

有关更多信息，请参见访问修饰符（C# 编程指南）。

可以声明具有类型参数的泛型选件类。有关更多信息，请参见 泛型选件类。

一个类可包含下列成员的声明：

- 构造函数
- 析构函数
- 常量
- 字段
- 方法

- 属性
- 索引器
- 运算符
- 事件
- 委托
- 类
- 接口
- 结构

下面的示例说明如何声明类的字段、构造函数和方法。该例还说明了如何实例化对象及如何打印实例数据。在此例中声明了两个类，一个是 Child 类，它包含两个私有字段（name 和 age）和两个公共方法。第二个类 StringTest 用来包含 Main。

```
class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }

    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}

class StringTest
{
    static void Main()
    {
        // Create objects by using the new operator:
        Child child1 = new Child("Craig", 11);
        Child child2 = new Child("Sally", 10);

        // Create an object using the default constructor:
        Child child3 = new Child();

        // Display results:
        Console.Write("Child #1: ");
        child1.PrintChild();
        Console.Write("Child #2: ");
        child2.PrintChild();
        Console.Write("Child #3: ");
        child3.PrintChild();
    }
}
/* Output:
    Child #1: Craig, 11 years old.
    Child #2: Sally, 10 years old.
    Child #3: N/A, 0 years old.
*/
```

注释

注意：在上例中，私有字段（name 和 age）只能通过 Child 类的公共方法访问。例如，不能在 Main 方法中使用如下语句打印 Child 的名称：

```
Console.Write(child1.name);   // Error
```

只有当 Child 是 Main 的成员时，才能从 Main 访问该类的私有成员。

类型声明在选件类中，不使用访问修饰符默认为 **private**，因此，在此示例中的数据成员会 **private**，如果移除了关键字。

最后要注意的是，默认情况下，对于使用默认构造函数 (child3) 创建的对象，age 字段初始化为零。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

引用类型（C# 参考）

委托（C# 参考）

委托类型的声明与方法签名相似，有一个返回值和任意数目任意类型的参数：

```
public delegate void TestDelegate(string message);
public delegate int TestDelegate(MyType m, long num);
```

delegate 是一种可用于封装命名或匿名方法的引用类型。委托类似于 C++ 中的函数指针；但是，委托是类型安全和可靠的。有关委托的应用，请参见委托和泛型委托。

备注

委托是事件的基础。

通过将委托与命名方法或匿名方法关联，可以实例化委托。有关更多信息，请参见命名方法和匿名方法。

必须使用具有兼容返回类型和输入参数的方法或 lambda 表达式实例化委托。有关方法签名中允许的差异程度的更多信息，请参见委托中的变体（C# 和 Visual Basic）。为了与匿名方法一起使用，委托和与之关联的代码必须一起声明。本节讨论这两种实例化委托的方法。

```
// Declare delegate -- defines required signature:
delegate double MathAction(double num);

class DelegateTest
{
    // Regular method that matches signature:
    static double Double(double input)
    {
        return input * 2;
    }

    static void Main()
    {
        // Instantiate delegate with named method:
        MathAction ma = Double;

        // Invoke delegate ma:
        double multByTwo = ma(4.5);
        Console.WriteLine("multByTwo: {0}", multByTwo);

        // Instantiate delegate with anonymous method:
        MathAction ma2 = delegate(double input)
        {
            return input * input;
        };

        double square = ma2(5);
        Console.WriteLine("square: {0}", square);

        // Instantiate delegate with lambda expression
        MathAction ma3 = s => s * s * s;
        double cube = ma3(4.375);

        Console.WriteLine("cube: {0}", cube);
    }
    // Output:
    // multByTwo: 9
    // square: 25
    // cube: 83.740234375
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

引用类型（C# 参考）

委托（C# 编程指南）

事件（C# 编程指南）

带有命名方法的委托与带有匿名方法的委托（C# 编程指南）

匿名方法（C# 编程指南）

dynamic（C# 参考）

在通过 **dynamic** 类型实现的操作中，该类型的作用是绕过编译时类型检查，改为在运行时解析这些操作。 **dynamic** 类型简化了对 COM API（例如 Office Automation API）、动态 API（例如 IronPython 库）和 HTML 文档对象模型 (DOM) 的访问。

在大多数情况下，**dynamic** 类型与 **object** 类型的行为是一样的。但是，不会用编译器对包含 **dynamic** 类型表达式的操作进行解析或类型检查。编译器将有关该操作信息打包在一起，并且该信息以后用于计算运行时操作。在此过程中，类型 **dynamic** 的变量会编译到类型 **object** 的变量中。因此，类型 **dynamic** 只在编译时存在，在运行时则不存在。

以下示例将类型为 **dynamic** 的变量与类型为 **object** 的变量对比。若要在编译时验证每个变量的类型，请将鼠标指针放在 **WriteLine** 语句中的 dyn 或 obj 上。IntelliSense 显示了 dyn 的“dynamic”和 obj 的“object”。

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

WriteLine 语句显示 dyn 和 obj 的运行时类型。此时，两者具有相同的整数类型。将生成以下输出：

System.Int32

System.Int32

若要查看 dyn 和 obj 之间的差异，请在前面示例的声明和 **WriteLine** 语句之间添加下列两行之间。

```
dyn = dyn + 3;
obj = obj + 3;
```

为尝试添加表达式 obj + 3 中的整数和对象报告编译器错误。但是，不会报告 dyn + 3 错误。编译时不会检查包含 dyn 的表达式，原因是 dyn 的类型为 **dynamic**。

上下文

dynamic 关键字可以直接出现或作为构造类型的组件在下列情况中出现：

- 在声明中，作为属性、字段、索引器、参数、返回值或类型约束的类型。下面的类定义在几个不同的声明中使用 **dynamic**。

```
class ExampleClass
{
    // A dynamic field.
    static dynamic field;

    // A dynamic property.
    dynamic prop { get; set; }

    // A dynamic return type and a dynamic parameter type.
    public dynamic exampleMethod(dynamic d)
    {
        // A dynamic local variable.
        dynamic local = "Local variable";
        int two = 2;

        if (d is int)
        {
            return local;
        }
        else
        {
            return two;
        }
    }
}
```

- 在显式类型转换中，作为转换的目标类型。

```
static void convertToDynamic()
{
    dynamic d;
    int i = 20;
    d = (dynamic)i;
    Console.WriteLine(d);

    string s = "Example string.";
    d = (dynamic)s;
    Console.WriteLine(d);

    DateTime dt = DateTime.Today;
    d = (dynamic)dt;
    Console.WriteLine(d);

}
// Results:
// 20
// Example string.
// 2/17/2009 9:12:00 AM
```

- 在以类型充当值（如 **is** 运算符或 **as** 运算符右侧）或者作为 **typeof** 的参数成为构造类型的一部分的任何上下文中。例如，可以在下列表达式中使用 **dynamic**。

```
int i = 8;
dynamic d;
// With the is operator.
// The dynamic type behaves like object. The following
// expression returns true unless someVar has the value null.
if (someVar is dynamic) { }

// With the as operator.
d = i as dynamic;

// With typeof, as part of a constructed type.
Console.WriteLine(typeof(List&lt;dynamic&gt;));

// The following statement causes a compiler error.
//Console.WriteLine(typeof(dynamic));
```

下面的示例以多个声明使用 **dynamic**。 Main 也用运行时类型检查对比编译时类型检查。

```
using System;

namespace DynamicExamples
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.exampleMethod(10));
            Console.WriteLine(ec.exampleMethod("value"));

            // The following line causes a compiler error because e
            // takes only one argument.
            //Console.WriteLine(ec.exampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.exampleMethod(10));

            // Because dynamic_ec is dynamic, the following call to
            // with two arguments does not produce an error at comp
            // However, itdoes cause a run-time error.
            //Console.WriteLine(dynamic_ec.exampleMethod(10, 4));
        }
    }

    class ExampleClass
    {
        static dynamic field;
        dynamic prop { get; set; }

        public dynamic exampleMethod(dynamic d)
        {
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}
// Results:
// Local variable
// 2
// Local variable
```

有关更多信息和示例，请参见使用类型 dynamic（C# 编程指南）。

请参阅

System.Dynamic.ExpandoObject

System.Dynamic.DynamicObject

使用类型 dynamic（C# 编程指南）

object（C# 参考）

is（C# 参考）

as（C# 参考）

typeof（C# 参考）

如何：使用 as 和 is 运算符安全地进行强制转换（C# 编程指南）

演练：创建和使用动态对象（C# 和 Visual Basic）

接口（C# 参考）

接口只包含方法、属性、事件或索引器的签名。 实现接口的类或结构必须实现接口定义中指定的接口成员。 在下面的示例，类 ImplementationClass 必须实现一个不具有参数并返回 **void** 的名为 SampleMethod 的方法。

有关更多信息和示例，请参见接口（C# 编程指南）。

示例

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

接口可以是命名空间或类的成员，并且可以包含下列成员的签名：

- 方法
- 属性
- 索引器
- 事件

一个接口可从一个或多个基接口继承。

当基类型列表包含基类和接口时，基类必须是列表中的第一项。

实现接口的类可以显式实现该接口的成员。 显式实现的成员不能通过类实例访问，而只能通过接口实例访问。

有关显式接口实现的更多详细信息和代码示例，请参见显式接口实现（C# 编程指南）。

示例

下面的示例演示了接口实现。 在此示例中，接口包含属性声明，类包含实现。 实现 IPoint 的类的任何实例都具有整数属性 x 和 y。

```
interface IPoint
{
   // Property signatures:
   int x
   {
      get;
      set;
   }

   int y
   {
      get;
      set;
   }
}

class Point : IPoint
{
   // Fields:
   private int _x;
   private int _y;

   // Constructor:
   public Point(int x, int y)
   {
      _x = x;
      _y = y;
   }

   // Property implementation:
   public int x
   {
      get
      {
         return _x;
      }

      set
      {
```

```
            _x = value;
        }
    }

    public int y
    {
        get
        {
            return _y;
        }
        set
        {
            _y = value;
        }
    }
}

class MainClass
{
    static void PrintPoint(IPoint p)
    {
        Console.WriteLine("x={0}, y={1}", p.x, p.y);
    }

    static void Main()
    {
        Point p = new Point(2, 3);
        Console.Write("My Point: ");
        PrintPoint(p);
    }
}
// Output: My Point: x=2, y=3
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

引用类型（C# 参考）

接口（C# 编程指南）

使用属性（C# 编程指南）

使用索引器（C# 编程指南）

class（C# 参考）

struct（C# 参考）

接口（C# 编程指南）

object（C# 参考）

object 类型在 .NET Framework 中是 Object 的别名。 在 C# 的统一类型系统中，所有类型（预定义类型、用户定义类型、引用类型和值类型）都是直接或间接从 Object 继承的。 可以将任何类型的值赋给 **object** 类型的变量。 将值类型的变量转换为对象的过程称为“装箱”。 将对象类型的变量转换为值类型的过程称为“取消装箱”。 有关更多信息，请参见装箱和取消装箱。

示例

下面的示例演示了 **object** 类型的变量如何接受任何数据类型的值，以及 **object** 类型的变量如何在 .NET Framework 中使用 Object 的方法。

```
class ObjectTest
{
   public int i = 10;
}

class MainClass2
{
   static void Main()
   {
      object a;
      a = 1;   // an example of boxing
      Console.WriteLine(a);
      Console.WriteLine(a.GetType());
      Console.WriteLine(a.ToString());

      a = new ObjectTest();
      ObjectTest classRef;
      classRef = (ObjectTest)a;
      Console.WriteLine(classRef.i);
   }
}
/* Output
    1
    System.Int32
    1
 * 10
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

引用类型（C# 参考）

值类型（C# 参考）

string（C# 参考）

string 类型表示一个字符序列（零个或更多 Unicode 字符）。 **string** 是 .NET Framework 中 String 的别名。

尽管 **string** 是引用类型，但定义相等运算符（**==** 和 **!=**）是为了比较 **string** 对象（而不是引用）的值。这使得对字符串相等性的测试更为直观。例如：

```
      string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine((object)a == (object)b);
```

这将先显示“True”，然后显示“False”，因为字符串的内容是相同的，但是 a 和 b 引用的不是同一个字符串实例。

- 运算符用于连接字符串：

```
string a = "good " + "morning";
```

这将创建一个包含“good morning”的字符串对象。

字符串是不可变的，即：字符串对象在创建后，尽管从语法上看您似乎可以更改其内容，但事实上并不可行。例如，编写此代码时，编译器实际上会创建一个新字符串对象来保存新的字符序列，且新对象将赋给 b。然后字符串“h”将适宜于垃圾回收。

```
      string b = "h";
b += "ello";
```

[] 运算符可以用于对 **string** 的各个字符的只读访问。

```
      string str = "test";
char x = str[2];  // x = 's';
```

字符串为 **string** 类型并可写成两种形式，即用引号引起来和用 @ 引起来。用引号引起来的字符串括在双引号 (") 内：

```
"good morning"  // a string literal
```

字符串文本可包含任何字符。包括转义序列。下面的示例使用转义序列 \ 来表示反斜杠，使用 \u0066 来表示字母 f，使用 \n 来表示换行符。

```
      string a = "\\\u0066\n";
Console.WriteLine(a);
```

注意
转义码 \u*dddd*（其中 *dddd* 是一个四位数）表示 Unicode 字符 U+*dddd*。此外还识别 8 位 Unicode 转义码：\U*dddddddd*。

原义字符串以 @ 开头并且也用双引号引起来。例如：

```
@"good morning"  // a string literal
```

原义字符串的优势在于不处理转义序列，因此很容易写入，例如完全限定的文件名就是原义字符串：

```
@"c:\Docs\Source\a.txt"  // rather than "c:\\Docs\\Source\\a.txt"
```

若要在一个用 @ 引起来的字符串中包括一个双引号，请使用两对双引号：

```
@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

@ 符号的另一种用法是使用作为 C# 关键字的被引用的 (/reference) 标识符。

有关 C# 中字符串的更多信息，请参见字符串（C# 编程指南）。

```
class SimpleStringTest
{
   static void Main()
   {
      string a = "\u0068ello ";
      string b = "world";
      Console.WriteLine( a + b );
      Console.WriteLine( a + b == "Hello World" ); // == performs a
   }
}
/* Output:
    hello world
    False
 */
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

在 .NET Framework 中使用字符串的最佳做法

C# 关键字

C# 编程指南

引用类型（C# 参考）

值类型（C# 参考）

.NET Framework 中的基本字符串操作

在 .NET Framework 中创建新字符串

设置数值结果表的格式（C# 参考）

内插字符串（C# 和 Visual Basic 引用）

用于构造字符串。内插字符串表达式类似于包含表达式的模板字符串。内插字符串表达式通过将包含的表达式替换为表达式结果的 ToString 表示形式来创建字符串。与复合格式设置相比，内插字符串在参数方面更易于理解。下面是内插字符串的示例：

```
Console.WriteLine($"Name = {name}, hours = {hours:hh}")
```

内插字符串的结构如下所示：

```
$ " <text> { <interpolation-expression> <optional-comma-field-widtl
```

可以在可使用字符串的任何位置使用内插字符串。当运行程序会执行带内插字符串的代码时，此代码会通过计算内插表达式来计算新的字符串。每次执行带内插字符串的代码时，都会发生此计算。

若要在内插字符串中包含大括号（“{”或“}”），请使用两个大括号，即“”。请参阅“隐式转换”部分以获取详细信息。

隐式转换

内插字符串中的隐式类型转换如下：

```
var s = $"hello, {name}"
System.IFormattable s = $"Hello, {name}"
System.FormattableString = $"Hello, {name}"
```

第一个示例生成一个 **string** 值，其中计算了所有字符串内插值。此为最终结果，并具有 string 类型。出现的所有双大括号（“”）都转换为单大括号。

第二个示例生成一个 IFormattable 变量，此变量允许使用固定上下文转换字符串。要使数值和数据格式在不同语言中正确无误，此操作非常有用。出现的所有双大括号（“”）仍保留为双大括号，直至将字符串设置为 ToString 格式。包含的所有内插表达式都转换为 {0}、{1} 等等。

```
s.ToString(null, System.Globalization.CultureInfo.InvariantCulture)
```

第三个示例生成一个 T;System.FormattableString，它允许检查由插值计算产生的对象。例如，检查对象及其呈现为字符串的方式可能有助于你在生成查询时防止注入攻击。借助 FormattableString，只需简单操作即可生成 InvariantCulture 和 CurrentCulture 字符串结果。在设置格式之前，出现在所有双大括号（“”）保留为双括号。包含的所有内插表达式都转换为 {0}、{1} 等等。

示例

```
$"Name = {name}, hours = {hours:hh}"
var s = $"hello, {name}"
System.IFormattable s = $"Hello, {name}"
System.FormattableString = $"Hello, {name}"
$"{person.Name, 20} is {person.Age:D3} year {(p.Age == 1 ? "" : "s'
```

无需在包含的内插表达式中引用引号字符，因为内插字符串表达式以 $ 开始，而且编译器会将包含的内插表达式扫描为平衡文本，直至它发现逗号、冒号或右大括号。出于相同原因，最后一个示例使用括号来使条件表达式 (p.Age == 1 ? "" : "s") 包含在其格式规范不以冒号开头的内插表达式中。在包含的内插表达式外部（但仍在内插字符串表达式之中），可按通常方式转义引号字符。

语法

```
expression:
    interpolated-string-expression

interpolated-string-expression:
    interpolated-string-start interpolations interpolated-string-en

interpolations:
    single-interpolation
    single-interpolation interpolated-string-mid interpolations

single-interpolation:
    interpolation-start
    interpolation-start : regular-string-literal

interpolation-start:
    expression
    expression , expression
```

语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

有关详细信息，请参阅 Visual Basic 语言参考。

请参阅

System.IFormattable

T;System.FormattableString

C# 参考

C# 编程指南

Visual Basic 语言参考

Visual Basic 编程指南

void（C# 参考）

当使用，因为方法的，**void** 返回类型指定方法不返回值。

void 在不允许参数列表方法。采用以下形式声明一个无参数的、不返回值的方法：

```
public void SampleMethod()
{
    // Body of the method.
}
```

也可在不安全上下文中使用 **void** 将指针声明为未知类型。有关更多信息，请参见指针类型（C# 编程指南）。

void 是 .NET Framework System.Void 类型的别名。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

引用类型（C# 参考）

值类型（C# 参考）

方法（C# 编程指南）

不安全代码和指针（C# 编程指南）

var（C# 参考）

从 Visual C# 3.0 开始，在方法范围中声明的变量可以具有隐式类型 **var**。隐式类型的本地变量是强类型变量（就好像您已经声明该类型一样），但由编译器确定类型。下面的两个 i 声明在功能上是等效的：

```
var i = 10; // implicitly typed
int i = 10; //explicitly typed
```

有关更多信息，请参见隐式类型的局部变量（C# 编程指南）和 Type Relationships in LINQ Query Operations (C#)。

下面的示例演示了两个查询表达式。在第一个表达式中，允许但不需要使用 **var**，因为可以将查询结果的类型显式声明为 **IEnumerable<string>**。但是，在第二个表达式中必须使用 **var**，因为结果是一个匿名类型集合，而该类型的名称只有编译器本身可以访问。请注意，在第二个示例中，**foreach** 迭代变量 item 也必须转换为隐式类型。

```
// Example #1: var is optional because
// the select clause specifies a string
string[] words = { "apple", "strawberry", "grape", "peach", "banana
var wordQuery = from word in words
                where word[0] == 'g'
                select word;

// Because each element in the sequence is a string,
// not an anonymous type, var is optional here also.
foreach (string s in wordQuery)
{
    Console.WriteLine(s);
}

// Example #2: var is required because
// the select clause specifies an anonymous type
var custQuery = from cust in customers
                where cust.City == "Phoenix"
                select new { cust.Name, cust.Phone };

// var must be used because each item
// in the sequence is an anonymous type
foreach (var item in custQuery)
{
    Console.WriteLine("Name={0}, Phone={1}", item.Name, item.Phone
}
```

请参阅

C# 参考

C# 编程指南

隐式类型的局部变量（C# 编程指南）

类型参考表（C# 参考）

以下参考表概括了 C# 的类型：

内置类型表

整型

浮点型

默认值

值类型

隐式数值转换

显式数值转换表

有关格式化数字类型输出的信息，请参见格式化数值结果表。

请参阅

C# 参考

C# 编程指南

引用类型（C# 参考）

值类型（C# 参考）

内置类型表（C# 参考）

下表显示了内置 C# 类型的关键字，这些类型是 System 命名空间中的预定义类型的别名。

C# 类型	.NET Framework 类型
bool	**System.Boolean**
byte	**System.Byte**
sbyte	**System.SByte**
char	**System.Char**
decimal	**System.Decimal**
double	**System.Double**
float	**System.Single**
int	**System.Int32**
uint	**System.UInt32**
long	**System.Int64**
ulong	**System.UInt64**
对象	**System.Object**
short	**System.Int16**
ushort	**System.UInt16**
string	**System.String**

备注

除了 **object** 和 **string** 外，表中的所有类型均称为简单类型。

C# 类型的关键字及其别名可以互换。例如，可使用下列两种声明中的一种来声明一个整数变量：

```
int x = 123;
System.Int32 x = 123;
```

若要显示任何 C# 类型的实际类型，请使用系统方法 GetType()。例如，下列语句显示了表示 myVariable 类型的系统别名：

```
Console.WriteLine(myVariable.GetType());
```

还可使用 typeof 运算符。

请参阅

C# 参考

C# 编程指南

C# 关键字

值类型（C# 参考）

默认值表（C# 参考）

设置数值结果表的格式（C# 参考）

dynamic（C# 参考）

类型参考表（C# 参考）

整型表（C# 参考）

下表显示了整型的大小和范围，这些类型构成了简单类型的一个子集。

类型	范围	大小
sbyte	-128 到 127	有符号 8 位整数
byte	0 到 255	无符号 8 位整数
char	U+0000 到 U+ffff	16 位 Unicode 字符
short	-32,768 到 32,767	有符号 16 位整数
ushort	0 到 65,535	无符号 16 位整数
int	-2,147,483,648 到 2,147,483,647	有符号 32 位整数
uint	0 到 4,294,967,295	无符号 32 位整数
long	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807	有符号 64 位整数
ulong	0 到 18,446,744,073,709,551,615	无符号 64 位整数

如果整数表示的值超出了 **ulong** 的范围，将产生编译错误。

请参阅

C# 参考

C# 编程指南

C# 关键字

内置类型表（C# 参考）

浮点型表（C# 参考）

默认值表（C# 参考）

设置数值结果表的格式（C# 参考）

类型参考表（C# 参考）

浮点型表（C# 参考）

下表显示了浮点型的精度和大致范围。

类型	大致范围	精度
float	±1.5e−45 到 ±3.4e38	7 位
double	±5.0e−324 到 ±1.7e308	15 到 16 位

请参阅

C# 参考

C# 编程指南

默认值表（C# 参考）

内置类型表（C# 参考）

整型表（C# 参考）

设置数值结果表的格式（C# 参考）

类型参考表（C# 参考）

decimal（C# 参考）

默认值表（C# 参考）

下表显示了由默认构造函数返回的值类型的默认值。默认构造函数是通过 **new** 运算符来调用的，如下所示：

```
int myInt = new int();
```

以上语句同下列语句效果相同：

```
int myInt = 0;
```

请记住：在 C# 中不允许使用未初始化的变量。

值类型	默认值
bool	**false**
byte	0
char	'\0'
decimal	0.0M
double	0.0D
enum	表达式 (E)0 产生的值，其中 E 为 enum 标识符。
float	0.0F
int	0
long	0L
sbyte	0
short	0
struct	将所有的值类型字段设置为默认值并将所有的引用类型字段设置为 **null** 时产生的值。
uint	0
ulong	0
ushort	0

请参阅

C# 参考

C# 编程指南

值类型表（C# 参考）

值类型（C# 参考）

内置类型表（C# 参考）

类型参考表（C# 参考）

值类型表（C# 参考）

下表按类别列出了 C# 的值类型。

值类型	类别	类型后缀
bool	Boolean	
byte	无符号、数值、整数	
char	无符号、数值、整数	
decimal	数值、十进制	M 或 m
double	数值、浮点	D 或 d
enum	Enumeration	
float	数值、浮点	F 或 f
int	有符号、数值、整数	
long	有符号、数值、整数	L 或 l
sbyte	有符号、数值、整数	
short	有符号、数值、整数	
struct	用户定义的结构	
uint	无符号、数值、整数	U 或 u
ulong	无符号、数值、整数	UL 或 ul
ushort	无符号、数值、整数	

请参阅

C# 参考

C# 编程指南

默认值表（C# 参考）

值类型（C# 参考）

设置数值结果表的格式（C# 参考）

类型参考表（C# 参考）

隐式数值转换表（C# 参考）

下表显示了预定义的隐式数值转换。隐式转换可能在多种情形下发生，包括调用方法时和在赋值语句中。

源	目标
sbyte	**short、int、long、float、double 或 decimal**
byte	**short、ushort、int、uint、long、ulong、float、double 或 decimal**
short	**int、long、float、double 或 decimal**
ushort	**int、uint、long、ulong、float、double 或 decimal**
int	**long、float、double 或 decimal**
uint	**long、ulong、float、double 或 decimal**
long	**float、double 或 decimal**
char	**ushort、int、uint、long、ulong、float、double 或 decimal**
float	**double**
ulong	**float、double 或 decimal**

备注

- 可能从转换中丢失精度，但不是模**int**， **uint**， **long**，或**ulong**到**float**和**long**或**ulong**到**double**。
- 不存在到 **char** 类型的隐式转换。
- 不存在浮点型与 **decimal** 类型之间的隐式转换。
- **int** 类型的常数表达式可转换为 **sbyte**、**byte**、**short**、**ushort**、**uint** 或 **ulong**，前提是常数表达式的值处于目标类型的范围之内。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

整型表（C# 参考）

内置类型表（C# 参考）

显式数值转换表（C# 参考）

强制转换和类型转换（C# 编程指南）

显式数值转换表（C# 参考）

显式数值转换用于通过显式转换表达式，将任何数字类型转换为任何其他数字类型。对于它不存在隐式转换。下表显示了这些转换。

有关转换的更多信息，请参见 强制转换和类型转换（C# 编程指南）。

源	目标
sbyte	**byte**、**ushort**、**uint**、**ulong** 或 **char**
byte	**Sbyte** 或 **char**
short	**sbyte**、**byte**、**ushort**、**uint**、**ulong** 或 **char**
ushort	**sbyte**、**byte**、**short** 或 **char**
int	**sbyte**、**byte**、**short**、**ushort**、**uint**、**ulong** 或 **char**
uint	**sbyte**、**byte**、**short**、**ushort**、**int** 或 **char**
long	**sbyte**、**byte**、**short**、**ushort**、**int**、**uint**、**ulong** 或 **char**
ulong	**sbyte**、**byte**、**short**、**ushort**、**int**、**uint**、**long** 或 **char**
char	**sbyte**、**byte** 或 **short**
float	**sbyte**、**byte**、**short**、**ushort**、**int**、**uint**、**long**、**ulong**、**char** 或 **decimal**
double	**sbyte**、**byte**、**short**、**ushort**、**int**、**uint**、**long**、**ulong**、**char**、**floa** 或 **decimal**
decimal	**sbyte**、**byte**、**short**、**ushort**、**int**、**uint**、**long**、**ulong**、**char**、**floa** 或 **double**

备注

- 显式数值转换可能导致精度损失或引发异常。
- 将 **decimal** 值转换为整型时，该值将舍入为与零最接近的整数值。如果结果整数值超出目标类型的范围，则会引发 OverflowException。
- 将 **double** 或 **float** 值转换为整型时，值会被截断。如果该结果整数值超出了目标值的范围，其结果将取决于溢出检查上下文。在 checked 上下文中，将引发 **OverflowException**；而在 unchecked 上下文中，结果将是一个未指定的目标类型的值。
- 将 **double** 转换为 **float** 时，**double** 值将舍入为最接近的 **float** 值。如果 **double** 值因过小或过大而使目标类型无法容纳它，则结果将为零或无穷大。

- 将 **float** 或 **double** 转换为 **decimal** 时，源值将转换为 **decimal** 表示形式，并舍入为第 28 个小数位之后最接近的数（如果需要）。根据源值的不同，可能产生以下结果：
  - 如果源值因过小而无法表示为 **decimal**，那么结果将为零。
  - 如果源值为 NaN（非数字值）、无穷大或因过大而无法表示为 **decimal**，则会引发 **OverflowException**。
- 将 **decimal** 转换为 **float** 或 **double** 时，**decimal** 值将舍入为最接近的 **double** 或 **float** 值。

有关显式转换的更多信息，请参见“C# 语言规范中的显式”。有关如何访问此规范的更多信息，请参见 C# 语言规范。

请参阅

C# 参考

C# 编程指南

强制转换和类型转换（C# 编程指南）

() 运算符（C# 参考）

整型表（C# 参考）

内置类型表（C# 参考）

隐式数值转换表（C# 参考）

设置数值结果表的格式（C# 参考）

通过使用 String.Format 方法，或者通过 Console.Write 或 Console.WriteLine 方法（这两个方法会调用 **String.Format**），可以设置数值结果的格式。通过使用格式字符串指定格式。下表包含受支持的标准格式字符串。格式字符串采用以下形式：*Axx*，其中 *A* 是格式说明符，*xx* 是精度说明符。格式说明符控制应用于数值的格式类型，而精度说明符则控制格式化输出的有效位数或小数位数。精度说明符的值介于 0 到 99。

有关标准和自定义格式字符串的更多信息，请参见 .NET Framework 中的格式化类型。有关 **String.Format** 方法的更多信息，请参见 String.Format。

格式说明符	描述	示例	**Output**
C 或 c	货币	Console.Write("{0:C}", 2.5);Console.Write("{0:C}", -2.5);	$2.50($2.50)
D 或 d	Decimal	Console.Write("{0:D5}", 25);	00025
E 或 e	科学型	Console.Write("{0:E}", 250000);	2.500000E+005
F 或 f	定点	Console.Write("{0:F2}", 25);Console.Write("{0:F0}", 25);	25.0025
G 或 g	常规	Console.Write("{0:G}", 2.5);	2.5
N 或 n	数字	Console.Write("{0:N}", 2500000);	2,500,000.00
X 或 x	十六进制	Console.Write("{0:X}", 250);Console.Write("{0:X}", 0xffff);	FAFFFF

请参阅

C# 参考

C# 编程指南

标准数字格式字符串

类型参考表（C# 参考）

string（C# 参考）

修饰符（C# 参考）

修饰符用于修改类型和类型成员的声明。本节介绍 C# 修饰符。

修饰符	用途
访问修饰符 public private internal protected	指定声明的类型和类型成员的可访问性。
abstract	指示某个类只能是其他类的基类。
async	指示修改后的方法、lambda 表达式或匿名方法是异步的。
const	指定无法修改字段或局部变量的值。
event	声明事件。
extern	指示在外部实现方法。
new	显式隐藏继承自基类的成员。
override	提供从基类继承的虚拟成员的新实现。
partial — 部分	在同一程序集中定义分部类、结构和方法。
readonly	声明一个字段，该字段只能赋值为该声明的一部分或者在同一个类的构造函数中。
sealed	指定无法继承类。
static	声明属于类型本身而不是特定对象的成员。
unsafe	声明不安全的上下文。
virtual	在派生类中声明其实现可由重写成员更改的方法或访问器。
volatile	指示字段可在程序中由操作系统、硬件或并发执行线程等项修改。

请参阅

C# 参考

C# 编程指南

C# 关键字

访问修饰符（C# 参考）

访问修饰符是一些关键字，用于指定声明的成员或类型的可访问性。本节介绍四个访问修饰符：

- public
- protected
- internal
- private

使用这些访问修饰符可指定下列五个可访问性级别：

public：访问不受限制。

protected：访问仅限于包含类或从包含类派生的类型。

Internal：访问仅限于当前程序集。

protected internal:访问限制到当前程序集或从包含派生的类型的类别。

private：访问仅限于包含类型。

本节还介绍以下内容：

- 可访问性级别：使用四个访问修饰符声明五个可访问性级别。
- 可访问域：指定在程序节的哪个位置可以引用成员。
- 可访问性级别的使用限制：概述了声明的可访问性级别的使用限制。

请参阅

C# 参考

C# 编程指南

C# 关键字

访问修饰符（C# 编程指南）

访问关键字（C# 参考）

修饰符（C# 参考）

可访问性级别（C# 参考）

使用访问修饰符 public、protected、internal 或 private 可以为成员指定以下声明的访问级别之一。

声明的可访问性	含义
public	访问不受限制。
protected	访问仅限于包含类或从包含类派生的类型。
internal	访问仅限于当前程序集。
protected internal	访问仅限于从包含类派生的当前程序集或类型。
private	访问仅限于包含类型。

一个成员或类型只能有一个访问修饰符，但使用 **protected****internal** 组合时除外。

命名空间上不允许使用访问修饰符。命名空间没有访问限制。

根据出现成员声明的上下文，只允许某些声明的可访问性。如果在成员声明中未指定访问修饰符，则使用默认的可访问性。

不嵌套在其他类型中的顶级类型的可访问性只能是 **internal** 或 **public**。这些类型的默认可访问性是 **internal**。

嵌套类型是其他类型的成员，它们可以具有下表所示的声明的可访问性。

属于	默认的成员可访问性	该成员允许的声明的可访问性
enum	**public**	无
class	**private**	**public****protected**internal**private****protected internal**
interface	**public**	无
struct	**private**	**public****internal****private**

嵌套类型的可访问性取决于它的可访问域，该域是由已声明的成员可访问性和直接包含类型的可访问域这二者共同确定的。但是，嵌套类型的可访问域不能超出包含类型的可访问域。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

访问修饰符（C# 参考）

可访问域（C# 参考）

可访问性级别的使用限制（C# 参考）

访问修饰符（C# 编程指南）

public（C# 参考）

private（C# 参考）

protected（C# 参考）

internal（C# 参考）

可访问域（C# 参考）

成员的可访问域指定程序中可以引用成员的部分。如果成员嵌套在其他类型中，其可访问域由该成员的可访问性级别和直接包含类型的可访问域共同确定。

顶级类型的可访问域至少是声明它的项目的程序文本。也就是说，域包括此项目的所有源文件。嵌套类型的可访问域至少是声明它的类型的程序文本，即域是一个类型体，包括所有嵌套的类型。嵌套类型的可访问域决不能超出包含类型的可访问域。这些概念在以下示例中加以说明。

该示例包含一个顶级类型 T1 和两个嵌套类 M1 和 M2。这两个类包含具有不同声明的可访问性的字段。在 Main 方法中，每个语句后都有注释，指示每个成员的可访问域。注意，尝试引用不可访问的成员的语句被注释掉了。如果希望查看由引用不可访问的成员所导致的编译器错误，请逐个移除注释。

```
namespace AccessibilityDomainNamespace
{
    public class T1
    {
        public static int publicInt;
        internal static int internalInt;
        private static int privateInt = 0;
        static T1()
        {
            // T1 can access public or internal members
            // in a public or private (or internal) nested class
            M1.publicInt = 1;
            M1.internalInt = 2;
            M2.publicInt = 3;
            M2.internalInt = 4;

            // Cannot access the private member privateInt
            // in either class:
            // M1.privateInt = 2; //CS0122
        }

        public class M1
        {
            public static int publicInt;
            internal static int internalInt;
            private static int privateInt = 0;
        }

        private class M2
        {
            public static int publicInt = 0;
            internal static int internalInt = 0;
            private static int privateInt = 0;
```

```
        }
    }

    class MainClass
    {
        static void Main()
        {
            // Access is unlimited:
            T1.publicInt = 1;

            // Accessible only in current assembly:
            T1.internalInt = 2;

            // Error CS0122: inaccessible outside T1:
            // T1.privateInt = 3;

            // Access is unlimited:
            T1.M1.publicInt = 1;

            // Accessible only in current assembly:
            T1.M1.internalInt = 2;

            // Error CS0122: inaccessible outside M1:
            //    T1.M1.privateInt = 3;

            // Error CS0122: inaccessible outside T1:
            //    T1.M2.publicInt = 1;

            // Error CS0122: inaccessible outside T1:
            //    T1.M2.internalInt = 2;

            // Error CS0122: inaccessible outside M2:
            //    T1.M2.privateInt = 3;

            // Keep the console open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();

        }
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

访问修饰符（C# 参考）

可访问性级别（C# 参考）

可访问性级别的使用限制（C# 参考）

访问修饰符（C# 编程指南）

public（C# 参考）

private（C# 参考）

protected（C# 参考）

internal（C# 参考）

可访问性级别的使用限制（C# 参考）

在一项声明中指定类型时，请检查类型的可访问性级别是否依赖成员或其他类型的可访问性级别。例如，直接基类必须至少与派生类具有同样的可访问性。以下声明导致编译器错误，因为基类 BaseClass 的可访问性小于 MyClass：

```
class BaseClass {...}
public class MyClass: BaseClass {...} // Error
```

下表汇总了对声明的可访问性级别的限制。

上下文	备注
类	类类型的直接基类必须至少与类类型本身具有同样的可访问性。
接口	接口类型的显式基接口必须至少与接口类型本身具有同样的可访问性。
委托	委托类型的返回类型和参数类型必须至少与委托类型本身具有同样的可访问性。
常量	常数的类型必须至少与常数本身具有同样的可访问性。
字段	字段的类型必须至少与字段本身具有同样的可访问性。
方法	方法的返回类型和参数类型必须至少与方法本身具有同样的可访问性。
属性	属性的类型必须至少与属性本身具有同样的可访问性。
事件	事件的类型必须至少与事件本身具有同样的可访问性。
索引器	索引器的类型和参数类型必须至少与索引器本身具有同样的可访问性。
运算符	运算符的返回类型和参数类型必须至少与运算符本身具有同样的可访问性。
构造函数	构造函数的参数类型必须至少与构造函数本身具有同样的可访问性。

下面的示例包含不同类型的错误声明。每个声明后的注释指示了预期的编译器错误。

```
// Restrictions on Using Accessibility Levels
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.
```

```
using System;

// A delegate:
delegate int MyDelegate();

class B
{
    // A private method:
    static int MyPrivateMethod()
    {
        return 0;
    }
}

public class A
{
    // Error: The type B is less accessible than the field A.myFiel
    public B myField = new B();

    // Error: The type B is less accessible
    // than the constant A.myConst.
    public readonly B myConst = new B();

    public B MyMethod()
    {
        // Error: The type B is less accessible
        // than the method A.MyMethod.
        return new B();
    }

    // Error: The type B is less accessible than the property A.MyP
    public B MyProp
    {
        set
        {
        }
    }

    MyDelegate d = new MyDelegate(B.MyPrivateMethod);
    // Even when B is declared public, you still get the error:
    // "The parameter B.MyPrivateMethod is not accessible due to
    // protection level."

    public static B operator +(A m1, B m2)
    {
        // Error: The type B is less accessible
        // than the operator A.operator +(A,B)
        return new B();
    }

    static void Main()
    {
        Console.Write("Compiled successfully");
```

```
        }
    }
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

访问修饰符（C# 参考）

可访问域（C# 参考）

可访问性级别（C# 参考）

访问修饰符（C# 编程指南）

public（C# 参考）

private（C# 参考）

protected（C# 参考）

internal（C# 参考）

internal（C# 参考）

internal 关键字是类型和类型的成员 访问修饰符。只有在同一程序集的文件中，内部类型或成员才是可访问的，如下例所示：

```
public class BaseClass
{
    // Only accessible within the same assembly
    internal static int x = 0;
}
```

从当前程序集或从包含类派生的类型，可以访问具有访问修饰符 **protected internal** 的类型或成员。

有关 **internal** 与其他访问修饰符的比较，请参见可访问性级别（C# 参考）和访问修饰符（C# 编程指南）。

有关程序集的更多信息，请参见程序集和全局程序集缓存（C# 和 Visual Basic）。

内部访问通常用于基于组件的开发，因为它使一组组件能够以私有方式进行合作，而不必向应用程序代码的其余部分公开。例如，用于生成图形用户界面的框架可以提供 Control 和 Form 类，这两个类通过使用具有内部访问权限的成员进行合作。由于这些成员是内部的，它们不向正在使用框架的代码公开。

从定义具有内部访问能力的类型或成员的程序集外部引用该类型或成员是错误的。

此示例包含两个文件：Assembly1.cs 和 Assembly1_a.cs。第一个文件包含内部基类 BaseClass。在第二个文件中，实例化 BaseClass 的尝试将产生错误。

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
   public static int intM = 0;
}
```

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
   static void Main()
   {
      BaseClass myBase = new BaseClass();   // CS0122
   }
}
```

在此示例中，使用与示例 1 中所用的文件相同的文件，并将 BaseClass 的可访问性级别更改为 **public**。还将成员 IntM 的可访问性级别更改为 **internal**。在此例中，您可以实例化类，但不能访问内部成员。

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
   internal static int intM = 0;
}
```

```
// Assembly2_a.cs
// Compile with: /reference:Assembly1.dll
public class TestAccess
{
   static void Main()
   {
      BaseClass myBase = new BaseClass();    // Ok.
      BaseClass.intM = 444;    // CS0117
   }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

访问修饰符（C# 参考）

可访问性级别（C# 参考）

修饰符（C# 参考）

public（C# 参考）

private（C# 参考）

protected（C# 参考）

private（C# 参考）

private 关键字是一个成员访问修饰符。 私有访问是允许的最低访问级别。 私有成员只有在声明它们的类和结构体中才是可访问的，如下例所示：

```
class Employee
{
    private int i;
    double d;   // private access by default
}
```

同一体中的嵌套类型也可以访问那些私有成员。

在定义私有成员的类或结构外引用它会导致编译时错误。

有关 **private** 与其他访问修饰符的比较，请参见可访问性级别（C# 参考）和访问修饰符（C# 编程指南）。

示例

在此示例中，Employee 类包含两个私有数据成员 name 和 salary。 作为私有成员，它们只能通过成员方法来访问。 添加名为 GetName 和 Salary 的公共方法，以便可以对私有成员进行受控的访问。 通过公共方法访问 name 成员，而通过公共只读属性访问salary 成员。 （有关更多信息，请参见 属性（C# 编程指南）。）

```
class Employee2
{
    private string name = "FirstName, LastName";
    private double salary = 100.0;

    public string GetName()
    {
        return name;
    }

    public double Salary
    {
        get { return salary; }
    }
}

class PrivateTest
{
    static void Main()
    {
        Employee2 e = new Employee2();

        // The data members are inaccessible (private), so
        // they can't be accessed like this:
        //    string n = e.name;
        //    double s = e.salary;

        // 'name' is indirectly accessed via method:
        string n = e.GetName();

        // 'salary' is indirectly accessed via property
        double s = e.Salary;
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

访问修饰符（C# 参考）

可访问性级别（C# 参考）

修饰符（C# 参考）

public（C# 参考）

protected（C# 参考）

internal（C# 参考）

protected（C# 参考）

protected 关键字是一个成员访问修饰符。受保护成员在其所在的类中可由派生类实例访问。有关 **protected** 与其他访问修饰符的比较，请参见可访问性级别。

只有在通过派生类类型发生访问时，基类的受保护成员在派生类中才是可访问的。例如，请看以下代码段：

```
class A
{
    protected int x = 123;
}

class B : A
{
    static void Main()
    {
        A a = new A();
        B b = new B();

        // Error CS1540, because x can only be accessed by
        // classes derived from A.
        // a.x = 10;

        // OK, because this class derives from A.
        b.x = 10;
    }
}
```

语句 a.x = 10 生成错误，因为它是在静态方法 Main 中生成的，而不是类 B 的实例。

结构成员无法受保护，因为无法继承结构。

此示例中，DerivedPoint 类派生自 Point。因此，可以从派生类直接访问基类的受保护成员。

```
class Point
{
    protected int x;
    protected int y;
}

class DerivedPoint: Point
{
    static void Main()
    {
        DerivedPoint dpoint = new DerivedPoint();

        // Direct access to protected members:
        dpoint.x = 10;
        dpoint.y = 15;
        Console.WriteLine("x = {0}, y = {1}", dpoint.x, dpoint.y);
    }
}
// Output: x = 10, y = 15
```

如果将 x 和 y 的访问级别更改为 private，编译器将发出错误信息：

'Point.y' is inaccessible due to its protection level.

'Point.x' is inaccessible due to its protection level.

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

访问修饰符（C# 参考）

可访问性级别（C# 参考）

修饰符（C# 参考）

public（C# 参考）

private（C# 参考）

internal（C# 参考）

public（C# 参考）

public 关键字是类型和类型成员的访问修饰符。公共访问是允许的最高访问级别。对访问公共成员没有限制，如下例所示：

```
class SampleClass
{
    public int x; // No access restrictions.
}
```

有关更多信息，请参见访问修饰符（C# 编程指南）和可访问性级别（C# 参考）。

在下面的示例中，声明了两个类：PointTest 和 MainClass。直接从 MainClass 访问 PointTest 的公共成员 x 和 y。

```
class PointTest
{
    public int x;
    public int y;
}

class MainClass4
{
    static void Main()
    {
        PointTest p = new PointTest();
        // Direct access to public members:
        p.x = 10;
        p.y = 15;
        Console.WriteLine("x = {0}, y = {1}", p.x, p.y);
    }
}
// Output: x = 10, y = 15
```

如果将 **public** 访问级别更改为 private 或 protected，您将收到错误信息：

“PointTest.y”不可访问，因为它受保护级别限制。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

访问修饰符（C# 编程指南）

C# 关键字

访问修饰符（C# 参考）

可访问性级别（C# 参考）

修饰符（C# 参考）

private（C# 参考）

protected（C# 参考）

internal（C# 参考）

abstract（C# 参考）

abstract 修饰符指示所修饰的内容缺少实现或未完全实现。abstract 修饰符可用于类、方法、属性、索引器和事件。在类声明中使用 **abstract** 修饰符以指示某个类只能是其他类的基类。标记为抽象或包含在抽象类中的成员必须通过从抽象类派生的类来实现。

在此例中，类 Square 必须提供 Area 的实现，因为它派生自 ShapesClass：

```
abstract class ShapesClass
{
    abstract public int Area();
}
class Square : ShapesClass
{
    int side = 0;

    public Square(int n)
    {
        side = n;
    }
    // Area method is required to avoid
    // a compile-time error.
    public override int Area()
    {
        return side * side;
    }

    static void Main()
    {
        Square sq = new Square(12);
        Console.WriteLine("Area of the square = {0}", sq.Area());
    }

    interface I
    {
        void M();
    }
    abstract class C : I
    {
        public abstract void M();
    }

}
// Output: Area of the square = 144
```

抽象类具有以下特性：

- 抽象类不能实例化。
- 抽象类可以包含抽象方法和抽象访问器。
- 不能用 sealed（C# 参考） 修饰符修饰抽象类，因为这两个修饰符的含义是相反的。采用 **sealed** 修饰符的类无法继承，而 **abstract** 修饰符要求对类进行继承。
- 从抽象类派生的非抽象类必须包括继承的所有抽象方法和抽象访问器的实际实现。

在方法或属性声明中使用 **abstract** 修饰符以指示方法或属性不包含实现。

抽象方法具有以下特性：

- 抽象方法是隐式的虚方法。
- 只允许在抽象类中使用抽象方法声明。
- 因为抽象方法声明不提供实际的实现，所以没有方法体；方法声明只是以一个分号结束，并且在签名后没有大括号 ({ })。例如：

  ```
  public abstract void MyMethod();
  ```

  实现由一个重写方法override（C# 参考）提供，此重写方法是非抽象类的一个成员。
- 在抽象方法声明中使用 static 或 virtual 修饰符是错误的。

除了在声明和调用语法上不同外，抽象属性的行为与抽象方法一样。

- 在静态属性上使用 **abstract** 修饰符是错误的。
- 在派生类中，通过包括使用 override 修饰符的属性声明，可以重写抽象的继承属性。

有关抽象类的更多信息，请参见抽象类、密封类及类成员（C# 编程指南）。

抽象类必须为所有接口成员提供实现。

实现接口的抽象类可以将接口方法映射到抽象方法上。例如：

```
interface I
{
    void M();
}
abstract class C : I
{
    public abstract void M();
}
```

在本例中，DerivedClass 类是从抽象类 BaseClass 派生的。抽象类包含一个抽象方法 AbstractMethod 和两个抽象属性 X 和 Y。

```
abstract class BaseClass   // Abstract class
{
    protected int _x = 100;
    protected int _y = 150;
    public abstract void AbstractMethod();   // Abstract method
    public abstract int X    { get; }
    public abstract int Y    { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }

    public override int X   // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y   // overriding property
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        DerivedClass o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine("x = {0}, y = {1}", o.X, o.Y);
    }
}
// Output: x = 111, y = 161
```

在上面的示例中，如果尝试通过使用下面的语句将抽象类实例化：

```
BaseClass bc = new BaseClass();   // Error
```

将出现错误，指出编译器无法创建抽象类“BaseClass”的实例。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

修饰符（C# 参考）

virtual（C# 参考）

override（C# 参考）

C# 关键字

async（C# 参考）

使用 **async** 修饰符可将方法、lambda 表达式或匿名方法指定为异步。如果对方法或表达式使用此修饰符，则其称为异步方法。

```
public async Task<int> ExampleMethodAsync()
{
    // . . . .
}
```

如果你不熟悉异步编程或不了解异步方法如何使用 **await** 关键字来完成可能需要长时间运行的工作而不阻止调用方的线程，则应阅读使用 Async 和 Await 的异步编程（C# 和 Visual Basic）中的简介。

```
string contents = await contentsTask;
```

方法将同步运行，直至到达其第一个 **await** 表达式，此时会将方法挂起，直到等待的任务完成。同时，如下节示例中所示，控件将返回到方法的调用方。

如果 **async** 关键字修改的方法不包含 **await** 表达式或语句，则该方法将同步执行。编译器警告将通知你不包含 **await** 的任何异步方法，因为该情况可能表示存在错误。请参见编译器警告（等级 1）CS4014。

async 关键字是上下文关键字，原因在于只有当它修饰方法、lambda 表达式或匿名方法时，它才是关键字。在所有其他上下文中，都会将其解释为标识符。

示例

下面的示例展示了异步事件处理程序 StartButton_Click 和异步方法 ExampleMethodAsync 之间的控制结构和流程。此异步方法得到的结果是一个下载网站的长度。此代码适用于在 Visual Studio 2013 中创建的 Windows Presentation Foundation (WPF) 应用或 Windows 应用商店应用；请参见有关设置应用的代码注释。

```
// You can run this code in Visual Studio 2013 as a WPF app or a Wi
// You need a button (StartButton) and a textbox (ResultsTextBox).
// Remember to set the names and handler so that you have something
// <Button Content="Button" HorizontalAlignment="Left" Margin="88,7
//         Click="StartButton_Click" Name="StartButton"/>
// <TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,
//          Text="TextBox" VerticalAlignment="Top" Width="310" Name

// To run the code as a WPF app:
//    paste this code into the MainWindow class in MainWindow.xaml.
//    add a reference to System.Net.Http, and
//    add a using directive for System.Net.Http.

// To run the code as a Windows Store app:
//    paste this code into the MainPage class in MainPage.xaml.cs,
//    add using directives for System.Net.Http and System.Threading

private async void StartButton_Click(object sender, RoutedEventArgs
{
    // ExampleMethodAsync returns a Task<int>, which means that the
    // eventually produces an int result. However, ExampleMethodAsy
    // the Task<int> value as soon as it reaches an await.
    ResultsTextBox.Text += "\n";
    try
    {
        int length = await ExampleMethodAsync();
        // Note that you could put "await ExampleMethodAsync()" in
        // "length" is, but due to when '+=' fetches the value of R
        // would not see the global side effect of ExampleMethodAsy
        ResultsTextBox.Text += String.Format("Length: {0}\n", lengt
    }
    catch (Exception)
    {
        // Process the exception if one occurs.
    }
}

public async Task<int> ExampleMethodAsync()
{
    var httpClient = new HttpClient();
    int exampleInt = (await httpClient.GetStringAsync("http://msdn.
    ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.
    // After the following return statement, any method that's awai
    // ExampleMethodAsync (in this case, StartButton_Click) can get
    // integer result.
    return exampleInt;
}
// Output:
// Preparing to finish ExampleMethodAsync.
// Length: 53292
```

◆重要事项
有关各项任务以及在等待任务期间所执行代码的更多信息，请参见使用 Async 和 Await 的异步编程（C# 和 Visual Basic）。有关使用类似元素的完整 WPF 示例，请参见演练：使用 Async 和 Await 访问 Web（C# 和 Visual Basic）。你可以从开发人员代码示例下载演练代码。

返回类型

异步方法的返回类型可以为 Task、Task<TResult> 或 void。方法不能声明任何 ref 或 out 参数，但是可以调用具有这类参数的方法。

如果异步方法的返回语句指定一个 **TResult** 类型的操作数，则你应指定 **Task<TResult>** 作为方法的返回类型。如果当方法完成时未返回有意义的值，则应使用 **Task**。即，对方法的调用将返回一个 **Task**，但是当 **Task** 完成时，任何等待 **Task** 的所有 **await** 表达式的计算结果都为 **void**。

你应主要使用 **void** 返回类型来定义事件处理程序，这些处理程序需要此返回类型。 **void** 返回异步方法的调用方不能等待，并且无法捕获该方法引发的异常。

有关更多信息和示例，请参见异步返回类型（C# 和 Visual Basic）。

请参阅

AsyncStateMachineAttribute

await（C# 参考）

演练：使用 Async 和 Await 访问 Web（C# 和 Visual Basic）

使用 Async 和 Await 的异步编程（C# 和 Visual Basic）

const（C# 参考）

使用 **const** 关键字来声明某个常量字段或常量局部变量。常量字段和常量局部变量不是变量并且不能修改。常量可以为数字、布尔值、字符串或 null 引用。不要创建常量来表示你需要随时更改的信息。例如，不要使用常量字段来存储服务的价格、产品版本号或公司的品牌名称。这些值会随着时间发生变化；因为编译器会传播常量，所以必须重新编译通过库编译的其他代码以查看更改。另请参阅 readonly 关键字。例如：

```
      const int x = 0;
public const double gravitationalConstant = 6.673e-11;
private const string productName = "Visual C#";
```

备注

常数声明的类型指定声明引入的成员类型。常量局部变量或常量字段的初始值设定项必须是一个可以隐式转换为目标类型的常量表达式。

常数表达式是在编译时可被完全计算的表达式。因此，对于引用类型的常数，可能的值只能是 **string** 和 null 引用。

常数声明可以声明多个常数，例如：

```
public const double x = 1.0, y = 2.0, z = 3.0;
```

不允许在常数声明中使用 **static** 修饰符。

常数可以参与常数表达式，如下所示：

```
public const int c1 = 5;
public const int c2 = c1 + 100;
```

注意

readonly 关键字与 **const** 关键字不同。 **const** 字段只能在该字段的声明中初始化。 **readonly** 字段可以在声明或构造函数中初始化。因此，根据所使用的构造函数，**readonly** 字段可能具有不同的值。另外，虽然 **const** 字段是编译时常量，但 **readonly** 字段可用于运行时常量，如此行所示：public static readonly uint l1 = (uint)DateTime.Now.Ticks;

```
public class ConstTest
{
    class SampleClass
    {
        public int x;
        public int y;
        public const int c1 = 5;
        public const int c2 = c1 + 5;

        public SampleClass(int p1, int p2)
        {
            x = p1;
            y = p2;
        }
    }

    static void Main()
    {
        SampleClass mC = new SampleClass(11, 22);
        Console.WriteLine("x = {0}, y = {1}", mC.x, mC.y);
        Console.WriteLine("c1 = {0}, c2 = {1}",
                          SampleClass.c1, SampleClass.c2 );
    }
}
/* Output
    x = 11, y = 22
    c1 = 5, c2 = 10
 */
```

此示例说明如何将常数用作局部变量。

```
public class SealedTest
{
    static void Main()
    {
        const int c = 707;
        Console.WriteLine("My local constant = {0}", c);
    }
}
// Output: My local constant = 707
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

修饰符（C# 参考）

readonly（C# 参考）

event（C# 参考）

event 关键字用于在发行者类中声明事件。

下面的示例演示如何声明和引发将 EventHandler 用作基础委托类型的事件。有关演示如何使用泛型 EventHandler<TEventArgs> 委托类型、如何订阅事件以及如何创建事件处理程序方法的完整代码示例，请参见如何：发布符合 .NET Framework 准则的事件（C# 编程指南）。

```
public class SampleEventArgs
{
    public SampleEventArgs(string s) { Text = s; }
    public String Text {get; private set;} // readonly
}
public class Publisher
{
    // Declare the delegate (if using non-generic pattern).
    public delegate void SampleEventHandler(object sender, SampleEv

    // Declare the event.
    public event SampleEventHandler SampleEvent;

    // Wrap the event in a protected virtual method
    // to enable derived classes to raise the event.
    protected virtual void RaiseSampleEvent()
    {
        // Raise the event by using the () operator.
        if (SampleEvent != null)
            SampleEvent(this, new SampleEventArgs("Hello"));
    }
}
```

事件是特殊类型的多路广播委托，仅可从声明它们的类或结构（发行者类）中调用。如果其他类或结构订阅了该事件，则当发行者类引发该事件时，会调用其事件处理程序方法。有关更多信息和代码示例，请参见事件（C# 编程指南） 和委托（C# 编程指南）。

事件可标记为 public、private、protected、internal 或 **protected****internal**。这些访问修饰符定义类的用户访问事件的方式。有关更多信息，请参见 访问修饰符（C# 编程指南）。

关键字和事件

下面的关键字可应用于事件。

关键字	说明	更多信息
static	即使类没有实例，调用方也能在任何时候使用该事件。	静态类和静态类成员（C# 编程指南）
virtual	允许派生类通过使用 override 关键字来重写事件行为。	继承（C# 编程指南）
sealed	指定对于派生类它不再属虚拟性质。	
abstract	编译器不会生成 **add** 和 **remove** 事件访问器块，因此派生类必须提供自己的实现。	

通过使用 static 关键字，可以将事件声明为静态事件。即使类没有任何实例，调用方也能在任何时候使用静态事件。有关更多信息，请参见 静态类和静态类成员（C# 编程指南）。

通过使用 virtual 关键字，可以将事件标记为虚拟事件。这样，派生类就可以通过使用 override 关键字来重写事件行为。有关更多信息，请参见 继承（C# 编程指南）。重写虚事件的事件也可以为 sealed，以表示其对于派生类不再是虚事件。最后，可以将事件声明为 abstract，这意味着编译器不会生成 **add** 和 **remove** 事件访问器块。因此派生类必须提供其自己的实现。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

add（C# 参考）

remove（C# 参考）

修饰符（C# 参考）

如何：合并委托（多路广播委托）（C# 编程指南）

extern（C# 参考）

extern 修饰符用于声明在外部实现的方法。 **extern** 修饰符的常见用法是在使用 Interop 服务调入非托管代码时与 **DllImport** 特性一起使用。在这种情况下，还必须将方法声明为 **static**，如下面的示例所示：

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

extern 关键字还可以定义外部程序集别名，使得可以从单个程序集中引用同一组件的不同版本。有关详细信息，请参阅 外部别名（C# 参考）。

将 abstract 和 **extern** 修饰符一起使用来修改同一成员是错误的做法。使用 **extern** 修饰符意味着方法是在 C# 代码的外部实现的，而使用 **abstract** 修饰符意味着类中未提供方法实现。

extern 关键字用于 C# 中时会比用于 C++ 中时受到更多的限制。若要比较 C# 关键字与 C++ 关键字，请参见 C++ 语言参考中的“使用 extern 指定链接”。

示例 1. 在此示例中，程序接收来自用户的字符串并将该字符串显示在消息框中。程序使用从 User32.dll 库导入的 MessageBox 方法。

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string

    static int Main()
    {
        string myString;
        Console.Write("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0)
    }

}
```

示例 2. 此示例阐释了调入 C 库（本机 DLL）的 C# 程序。

1.创建以下 C 文件并将其命名为 cmdll.c：

```
// cmdll.c
// Compile with: /LD
int __declspec(dllexport) SampleMethod(int i)
{
   return i*10;
}
```

2.从 Visual Studio 安装目录打开 Visual Studio x64（或 x32）本机工具命令提示符窗口，并通过在命令提示符处键入 **cl /LD cmdll.c** 来编译 cmdll.c 文件。

3.在相同的目录中，创建以下 C# 文件并将其命名为 cm.cs：

```
// cm.cs
using System;
using System.Runtime.InteropServices;
public class MainClass
{
   [DllImport("Cmdll.dll")]
   public static extern int SampleMethod(int x);

   static void Main()
   {
      Console.WriteLine("SampleMethod() returns {0}.", SampleMethoc
   }
}
```

3.从 Visual Studio 安装目录打开一个 Visual Studio x64（或 x32）本机工具命令提示符窗口，并通过键入以下内容来编译 cm.cs 文件：

> **csc cm.cs**（针对 x64 命令提示符）
>
> - 或 -
>
> **csc /platform:x86 cm.cs**（针对 x32 命令提示符）

这将创建可执行文件 cm.exe。

4.运行 cm.exe。 SampleMethod 方法将值 5 传递到 DLL 文件，这将返回该值与 10 相乘后的结果。程序生成以下输出：

```
SampleMethod() returns 50.
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

System.Runtime.InteropServices.DllImportAttribute

C# 参考

C# 编程指南

C# 关键字

修饰符（C# 参考）

in（泛型修饰符）（C# 参考）

对于泛型类型参数，**in** 关键字指定该类型参数是逆变的。可以在泛型接口和委托中使用 **in** 关键字。

通过逆变，可以使用与泛型参数指定的派生类型相比，派生程度更小的类型。这样可以对委托类型和实现变体接口的类进行隐式转换。引用类型支持泛型类型参数中的协变和逆变，但值类型不支持。

在泛型接口或委托中，如果类型形参仅用作方法返回类型，而不用于方法实参，则可声明为协变的。 **Ref** 和 **out** 参数不能为变体。

如果接口具有逆变类型形参，则允许其方法接受与接口类型形参指定的派生类型相比，派生程度更小的类型的实参。例如，由于在 .NET Framework 4 的 IComparer<T> 接口中，类型 T 是逆变的，因此如果 Employee 继承 Person，则无需使用任何特殊转换方法，就可以将 IComparer(Of Person) 类型的对象指派给 IComparer(Of Employee) 类型的对象。

可以向逆变委托分配同一类型的其他委托，但需使用派生程度较小的泛型类型参数。

有关更多信息，请参见协变和逆变（C# 和 Visual Basic）。

下例演示如何声明、扩展和实现一个逆变泛型接口。此外还演示了如何对实现此接口的类使用隐式转换。

```
// Contravariant interface.
interface IContravariant<in A> { }

// Extending contravariant interface.
interface IExtContravariant<in A> : IContravariant<A> { }

// Implementing contravariant interface.
class Sample<A> : IContravariant<A> { }

class Program
{
    static void Test()
    {
        IContravariant<Object> iobj = new Sample<Object>();
        IContravariant<String> istr = new Sample<String>();

        // You can assign iobj to istr because
        // the IContravariant interface is contravariant.
        istr = iobj;
    }
}
```

下例演示如何声明、实例化和调用一个逆变泛型委托。此外还演示了如何隐式转换委托类型。

```
// Contravariant delegate.
public delegate void DContravariant<in A>(A argument);

// Methods that match the delegate signature.
public static void SampleControl(Control control)
{ }
public static void SampleButton(Button button)
{ }

public void Test()
{

    // Instantiating the delegates with the methods.
    DContravariant<Control> dControl = SampleControl;
    DContravariant<Button> dButton = SampleButton;

    // You can assign dControl to dButton
    // because the DContravariant delegate is contravariant.
    dButton = dControl;

    // Invoke the delegate.
    dButton(new Button());
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

out（泛型修饰符）（C# 参考）

协变和逆变（C# 和 Visual Basic）

修饰符（C# 参考）

out（泛型修饰符）（C# 参考）

对于泛型类型参数，**out** 关键字指定该类型参数是协变的。可以在泛型接口和委托中使用 **out** 关键字。

通过协变，可以使用与泛型参数指定的派生类型相比，派生程度更大的类型。这样可以对委托类型和实现变体接口的类进行隐式转换。引用类型支持协变和逆变，但值类型不支持。

如果接口具有协变类型形参，则允许其方法返回与接口类型形参指定的派生类型相比，派生程度更大的类型的实参。例如，由于在 .NET Framework 4 的 IEnumerable<T> 接口中，类型 T 是协变的，因此无需使用任何特殊转换方法就可以将 IEnumerabe(Of String) 类型的对象分配给 IEnumerable(Of Object) 类型的对象。

可以向协变委托分配同一类型的其他委托，但需使用派生程度较大的泛型类型参数。

有关更多信息，请参见协变和逆变（C# 和 Visual Basic）。

下例演示如何声明、扩展和实现一个协变泛型接口。此外还演示了如何对实现协变接口的类使用隐式转换。

```
// Covariant interface.
interface ICovariant<out R> { }

// Extending covariant interface.
interface IExtCovariant<out R> : ICovariant<R> { }

// Implementing covariant interface.
class Sample<R> : ICovariant<R> { }

class Program
{
    static void Test()
    {
        ICovariant<Object> iobj = new Sample<Object>();
        ICovariant<String> istr = new Sample<String>();

        // You can assign istr to iobj because
        // the ICovariant interface is covariant.
        iobj = istr;
    }
}
```

在泛型接口中，当符合下列条件时，可以将类型参数声明为是协变的。

- 类型形参仅用作接口方法的返回类型，不用作方法实参的类型。

> 注意
>
> 此规则有一个例外。如果在协变接口中，包含用作方法参数的逆变泛型委托，则可以将协变类型用作此委托的泛型类型参数。有关协变和逆变泛型委托的更多信息，请参见委托中的变体（C# 和 Visual Basic）和对 Func 和 Action 泛型委托使用变体（C# 和 Visual Basic）。

- 类型参数不用作接口方法的泛型约束。

下例演示如何声明、实例化和调用一个协变泛型委托。此外还演示了如何隐式转换委托类型。

```
// Covariant delegate.
public delegate R DCovariant<out R>();

// Methods that match the delegate signature.
public static Control SampleControl()
{ return new Control(); }

public static Button SampleButton()
{ return new Button(); }

public void Test()
{
    // Instantiate the delegates with the methods.
    DCovariant<Control> dControl = SampleControl;
    DCovariant<Button> dButton = SampleButton;

    // You can assign dButton to dControl
    // because the DCovariant delegate is covariant.
    dControl = dButton;

    // Invoke the delegate.
    dControl();
}
```

在泛型委托中，如果类型仅用作方法返回类型，且不用于方法参数，则可声明为是协变的。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

泛型接口中的变体（C# 和 Visual Basic）

in（泛型修饰符）（C# 参考）

修饰符（C# 参考）

override（C# 参考）

要扩展或修改继承的方法、属性、索引器或事件的抽象实现或虚实现，必须使用 **override** 修饰符。

在此示例中，Square 类必须提供 Area 的重写实现，因为 Area 继承自抽象的 ShapesClass：

```
abstract class ShapesClass
{
    abstract public int Area();
}
class Square : ShapesClass
{
    int side = 0;

    public Square(int n)
    {
        side = n;
    }
    // Area method is required to avoid
    // a compile-time error.
    public override int Area()
    {
        return side * side;
    }

    static void Main()
    {
        Square sq = new Square(12);
        Console.WriteLine("Area of the square = {0}", sq.Area());
    }

    interface I
    {
        void M();
    }
    abstract class C : I
    {
        public abstract void M();
    }

}
// Output: Area of the square = 144
```

override 方法提供从基类继承的成员的新实现。由 **override** 声明重写的方法称为重写基方法。重写的基方法必须与 **override** 方法具有相同的签名。有关继承的信息，请参见继承（C# 编程指南）。

不能重写非虚方法或静态方法。重写的基方法必须是 **virtual**、**abstract** 或 **override** 的。

override 声明不能更改 **virtual** 方法的可访问性。 **override** 方法和 **virtual** 方法必须具有相同的访问级别修饰符。

您不能使用 **new**、**static** 或 **virtual** 修饰符来修改 **override** 方法。

重写属性声明必须指定与继承属性完全相同的访问修饰符、类型和名称，并且被重写的属性必须是 **virtual**、**abstract** 或 **override** 的。

有关如何使用 **override** 关键字的更多信息，请参见使用 Override 和 New 关键字进行版本控制（C# 编程指南）和了解何时使用 Override 和 New 关键字。

此示例定义了一个名为 Employee 的基类和一个名为 SalesEmployee 的派生类。SalesEmployee 类包括一个额外的属性 salesbonus，并重写方法 CalculatePay 以便将该属性考虑在内。

```
class TestOverride
{
    public class Employee
    {
        public string name;

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derrived classes.
        protected decimal basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            this.name = name;
            this.basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal salesbonus;
```

```
        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
        public SalesEmployee(string name, decimal basepay,
                  decimal salesbonus) : base(name, basepay)
        {
            this.salesbonus = salesbonus;
        }

        // Override the CalculatePay method
        // to take bonus into account.
        public override decimal CalculatePay()
        {
            return basepay + salesbonus;
        }
    }

    static void Main()
    {
        // Create some new employees.
        SalesEmployee employee1 = new SalesEmployee("Alice",
                      1000, 500);
        Employee employee2 = new Employee("Bob", 1200);

        Console.WriteLine("Employee4 " + employee1.name +
                  " earned: " + employee1.CalculatePay());
        Console.WriteLine("Employee4 " + employee2.name +
                  " earned: " + employee2.CalculatePay());
    }
}
/*
    Output:
    Employee4 Alice earned: 1500
    Employee4 Bob earned: 1200
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

继承（C# 编程指南）

C# 关键字

修饰符（C# 参考）

abstract（C# 参考）

virtual（C# 参考）

new（C# 参考）

多态性（C# 编程指南）

readonly（C# 参考）

readonly 关键字是可以在字段上使用的修饰符。当字段声明包括 **readonly** 修饰符时，该声明引入的字段赋值只能作为声明的一部分出现，或者出现在同一类的构造函数中。

在此示例中，字段 year 的值无法在 ChangeYear 方法中更改，即使在类构造函数中给它赋了值。

```
class Age
{
    readonly int _year;
    Age(int year)
    {
        _year = year;
    }
    void ChangeYear()
    {
        //_year = 1967; // Compile error if uncommented.
    }
}
```

只能在下列上下文中对 **readonly** 字段进行赋值：

- 当在声明中初始化变量时，例如：

  ```
  public readonly int y = 5;
  ```

- 对于实例字段，在包含字段声明的类的实例构造函数中；或者，对于静态字段，在包含字段声明的类的静态构造函数中。也只有在这些上下文中，将 readonly 字段作为 out 或 ref 参数传递才有效。

> 注意
>
> **readonly** 关键字与 const 关键字不同。 **const** 字段只能在该字段的声明中初始化。 **readonly** 字段可以在声明或构造函数中初始化。因此，根据所使用的构造函数，**readonly** 字段可能具有不同的值。另外，**const** 字段为编译时常数，而 **readonly** 字段可用于运行时常数，如下例所示：

```
public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
```

```
public class ReadOnlyTest
{
   class SampleClass
   {
      public int x;
      // Initialize a readonly field
      public readonly int y = 25;
      public readonly int z;

      public SampleClass()
      {
         // Initialize a readonly instance field
         z = 24;
      }

      public SampleClass(int p1, int p2, int p3)
      {
         x = p1;
         y = p2;
         z = p3;
      }
   }

   static void Main()
   {
      SampleClass p1 = new SampleClass(11, 21, 32);   // OK
      Console.WriteLine("p1: x={0}, y={1}, z={2}", p1.x, p1.y, p1.z
      SampleClass p2 = new SampleClass();
      p2.x = 55;   // OK
      Console.WriteLine("p2: x={0}, y={1}, z={2}", p2.x, p2.y, p2.z
   }
}
/*
 Output:
    p1: x=11, y=21, z=32
    p2: x=55, y=25, z=24
*/
```

在前面的示例中，如果使用这样的语句：

p2.y = 66; // Error

将收到编译器错误信息：

The left-hand side of an assignment must be an l-value

这与尝试将值赋给常数时收到的错误相同。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

修饰符（C# 参考）

const（C# 参考）

字段（C# 编程指南）

sealed（C# 参考）

当对一个类应用 **sealed** 修饰符时，此修饰符会阻止其他类从该类继承。在下面的示例中，类 B 从类 A 继承，但是任何类都不能从类 B 继承。

```
class A {}
sealed class B : A {}
```

还可以在重写基类中的虚方法或虚属性的方法或属性上使用 **sealed** 修饰符。这将使您能够允许类从您的类继承，并防止它们重写特定的虚方法或虚属性。

在下面的示例中，Z 从 Y 继承，但 Z 无法重写在 X 中声明并在 Y 中密封的虚函数 F。

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}
class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F");
    protected override void F2() { Console.WriteLine("Y.F2"); }
}
class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("C.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

当在类中定义新的方法或属性时，通过不将这些方法或属性声明为 virtual，可防止派生类重写这些方法或属性。

将 abstract 修饰符用于密封类是错误的做法，因为抽象类必须由提供抽象方法或属性的实现的类继承。

当应用于方法或属性时，**sealed** 修饰符必须始终与 override 一起使用。

由于结构是隐式密封的，因此它们不能被继承。

有关更多信息，请参见 继承（C# 编程指南）。

有关更多示例，请参见抽象类、密封类及类成员（C# 编程指南）。

```
sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        SealedClass sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine("x = {0}, y = {1}", sc.x, sc.y);
    }
}
// Output: x = 110, y = 150
```

在上一个示例中，您可能尝试使用下面的语句从密封类继承：

class MyDerivedC: SealedClass {} // Error

将产生一条错误消息：

'MyDerivedC' cannot inherit from sealed class 'SealedClass'.

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

备注

若要确定是否密封类、方法或属性，通常应考虑以下两点：

- 派生类利用自定义类的功能所获得的可能好处。
- 派生类在修改类之后导致其无法正常工作或按预期工作的可能性。

请参阅

C# 参考

C# 编程指南

C# 关键字

静态类和静态类成员（C# 编程指南）

抽象类、密封类及类成员（C# 编程指南）

访问修饰符（C# 编程指南）

修饰符（C# 参考）

override（C# 参考）

virtual（C# 参考）

static（C# 参考）

使用 **static** 修饰符声明属于类型本身而不是属于特定对象的静态成员。 **static** 修饰符可用于类、字段、方法、属性、运算符、事件和构造函数，但不能用于索引器、析构函数或类以外的类型。有关更多信息，请参见 静态类和静态类成员（C# 编程指南）。

下面的类声明为 **static**，并且只包含 **static** 方法：

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/  }
}
```

常数或者类型声明隐式地是静态成员。

不能通过实例引用静态成员。然而，可以通过类型名称引用它。例如，请考虑以下类：

```
public class MyBaseC
{
    public struct MyStruct
    {
        public static int x = 100;
    }
}
```

若要引用静态成员 x，请使用完全限定名 MyBaseC.MyStruct.x，除非可从相同范围访问成员：

```
Console.WriteLine(MyBaseC.MyStruct.x);
```

尽管类的实例包含该类所有实例字段的单独副本，但每个静态字段只有一个副本。

不可以使用 this 来引用静态方法或属性访问器。

如果对类应用 **static** 关键字，则该类的所有成员都必须是静态的。

类和静态类可以有静态构造函数。静态构造函数在程序开始和类实例化之间的某个时刻调用。

> 📝 注意
>
> **static** 关键字在使用上比在 C++ 中有更多限制。若要与 C++ 关键字比较，请参见 Static (C++)。

为了说明静态成员，请看一个表示公司雇员的类。假设该类包含一种对雇员计数的方法和一个存储雇员数的字段。该方法和字段都不属于任何实例雇员，而是属于公司类。因此，应该将它们声明为此类的静态成员。

此示例读取新雇员的姓名和 ID，将雇员计数器加一，并显示新雇员的信息和新的雇员数。为简单起见，该程序从键盘读取当前的雇员数。在实际的应用中，应从文件读取此信息。

```
    public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object:
        Employee4 e = new Employee4(name, id);
        Console.Write("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();
```

```
        // Display the new information:
        Console.WriteLine("Name: {0}", e.name);
        Console.WriteLine("ID:   {0}", e.id);
        Console.WriteLine("New Number of Employees: {0}",
                      Employee4.employeeCounter);
    }
}
    /*
    Input:
    Matthias Berndt
    AF643G
    15
     *
    Sample Output:
    Enter the employee's name: Matthias Berndt
    Enter the employee's ID: AF643G
    Enter the current number of employees: 15
    Name: Matthias Berndt
    ID:   AF643G
    New Number of Employees: 16
    */
```

此示例说明：虽然可以用另一个尚未声明的静态字段实例化一个静态字段，但直到向后者显式赋值后，才能确定结果。

```
class Test
{
   static int x = y;
   static int y = 5;

   static void Main()
   {
      Console.WriteLine(Test.x);
      Console.WriteLine(Test.y);

      Test.x = 99;
      Console.WriteLine(Test.x);
   }
}
/*
Output:
    0
    5
    99
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

修饰符（C# 参考）

静态类和静态类成员（C# 编程指南）

unsafe（C# 参考）

unsafe 关键字表示不安全上下文，该上下文是任何涉及指针的操作所必需的。有关更多信息，请参见 不安全代码和指针（C# 编程指南）。

可以在类型或成员的声明中使用 **unsafe** 修饰符。因此，类型或成员的整个正文范围均被视为不安全上下文。例如，以下是用 **unsafe** 修饰符声明的方法：

```
        unsafe static void FastCopy(byte[] src, byte[] dst, int count
{
    // Unsafe context: can use pointers here.
}
```

不安全上下文的范围从参数列表扩展到方法的结尾，因此指针在以下参数列表中也可以使用：

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...]
```

还可以使用不安全块从而能够使用该块内的不安全代码。例如：

```
        unsafe
{
    // Unsafe context: can use pointers here.
}
```

若要编译不安全代码，必须指定 /unsafe 编译器选项。无法通过公共语言运行时验证不安全代码。

```
// compile with: /unsafe

class UnsafeTest
{
   // Unsafe method: takes pointer to int:
   unsafe static void SquarePtrParam(int* p)
   {
      *p *= *p;
   }

   unsafe static void Main()
   {
      int i = 5;
      // Unsafe method: uses address-of operator (&):
      SquarePtrParam(&i);
      Console.WriteLine(i);
   }
}
// Output: 25
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

fixed 语句（C# 参考）

不安全代码和指针（C# 编程指南）

固定大小的缓冲区（C# 编程指南）

virtual（C# 参考）

virtual 关键字用于修饰方法、属性、索引器或事件声明，并使它们可以在派生类中被重写。例如，此方法可被任何继承它的类重写。

```
public virtual double Area()
{
    return x * y;
}
```

虚拟成员的实现可由派生类中的重写成员更改。有关如何使用 **virtual** 关键字的更多信息，请参见使用 Override 和 New 关键字进行版本控制（C# 编程指南）和了解何时使用 Override 和 New 关键字（C# 编程指南）。

备注

调用虚方法时，将为重写成员检查该对象的运行时类型。将调用大部分派生类中的该重写成员，如果没有派生类重写该成员，则它可能是原始成员。

默认情况下，方法是非虚拟的。不能重写非虚方法。

virtual 修饰符不能与 **static**、**abstract, private** 或 **override** 修饰符一起使用。下面的示例演示一个虚拟属性：

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set a
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string name;

   // Override auto-implemented property with ordinary property
   // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (value != String.Empty)
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }

}
```

除了声明和调用语法不同外，虚拟属性的行为与抽象方法一样。

- 在静态属性上使用 **virtual** 修饰符是错误的。
- 通过包括使用 **override** 修饰符的属性声明，可在派生类中重写虚拟继承属性。

在该示例中，Shape 类包含 x、y 两个坐标和 Area() 虚方法。不同的形状类，如 Circle、Cylinder 和 Sphere 继承 Shape 类，并为每个图形计算表面积。每个派生类都有各自的 Area() 重写实现。

通知继承的类 Circle、 Sphere和 Cylinder 初始化基类的所有使用构造函数，如以下声明所示。

```
public Cylinder(double r, double h): base(r, h) {}
```

下面的过程通过调用 Area() 方法的适当实现计算并显示由每个图形的合适区域，根据与该方法的对象。

```
class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double x, y;
        public Shape()
        {
        }
        public Shape(double x, double y)
        {
            this.x = x;
            this.y = y;
        }

        public virtual double Area()
        {
            return x * y;
        }
    }

    public class Circle : Shape
    {
        public Circle(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return PI * x * x;
        }
    }

    class Sphere : Shape
    {
        public Sphere(double r) : base(r, 0)
        {
        }
```

```
        public override double Area()
        {
            return 4 * PI * x * x;
        }
    }

    class Cylinder : Shape
    {
        public Cylinder(double r, double h) : base(r, h)
        {
        }

        public override double Area()
        {
            return 2 * PI * x * x + 2 * PI * x * y;
        }
    }

    static void Main()
    {
        double r = 3.0, h = 5.0;
        Shape c = new Circle(r);
        Shape s = new Sphere(r);
        Shape l = new Cylinder(r, h);
        // Display results:
        Console.WriteLine("Area of Circle   = {0:F2}", c.Area());
        Console.WriteLine("Area of Sphere   = {0:F2}", s.Area());
        Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
        }
    }
    /*
        Output:
        Area of Circle   = 28.27
        Area of Sphere   = 113.10
        Area of Cylinder = 150.80
    */
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

修饰符（C# 参考）

C# 关键字

多态性（C# 编程指南）

abstract（C# 参考）

override（C# 参考）

new（C# 参考）

volatile（C# 参考）

volatile 关键字指示一个字段可以由多个同时执行的线程修改。声明为 **volatile** 的字段不受编译器优化（假定由单个线程访问）的限制。这样可以确保该字段在任何时间呈现的都是最新的值。

volatile 修饰符通常用于由多个线程访问但不使用 lock 语句对访问进行序列化的字段。

volatile 关键字可应用于以下类型的字段：

- 引用类型。
- 指针类型（在不安全的上下文中）。请注意，虽然指针本身可以是可变的，但是它指向的对象不能是可变的。换句话说，您无法声明“指向可变对象的指针”。
- 类型，如 sbyte、byte、short、ushort、int、uint、char、float 和 bool。
- 具有以下基类型之一的枚举类型：byte、sbyte、short、ushort、int 或 uint。
- 已知为引用类型的泛型类型参数。
- IntPtr 和 UIntPtr。

可变关键字仅可应用于类或结构字段。不能将局部变量声明为 **volatile**。

下面的示例说明如何将公共字段变量声明为 **volatile**。

```
class VolatileTest
{
    public volatile int i;

    public void Test(int _i)
    {
        i = _i;
    }
}
```

下面的示例演示如何创建辅助线程，并用它与主线程并行执行处理。有关多线程处理的背景信息，请参见Managed Threading和线程处理（C# 和 Visual Basic）。

```
using System;
using System.Threading;

public class Worker
{
    // This method is called when the thread is started.
```

```
    public void DoWork()
    {
        while (!_shouldStop)
        {
            Console.WriteLine("Worker thread: working...");
        }
        Console.WriteLine("Worker thread: terminating gracefully.")
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    // Keyword volatile is used as a hint to the compiler that this
    // member is accessed by multiple threads.
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    static void Main()
    {
        // Create the worker thread object. This does not start the
        Worker workerObject = new Worker();
        Thread workerThread = new Thread(workerObject.DoWork);

        // Start the worker thread.
        workerThread.Start();
        Console.WriteLine("Main thread: starting worker thread...")

        // Loop until the worker thread activates.
        while (!workerThread.IsAlive) ;

        // Put the main thread to sleep for 1 millisecond to
        // allow the worker thread to do some work.
        Thread.Sleep(1);

        // Request that the worker thread stop itself.
        workerObject.RequestStop();

        // Use the Thread.Join method to block the current thread
        // until the object's thread terminates.
        workerThread.Join();
        Console.WriteLine("Main thread: worker thread has terminate
    }
    // Sample output:
    // Main thread: starting worker thread...
    // Worker thread: working...
    // Worker thread: working...
    // Worker thread: working...
    // Worker thread: working...
    // Worker thread: working...
    // Worker thread: working...
    // Worker thread: terminating gracefully.
```

```
        // Main thread: worker thread has terminated.
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

修饰符（C# 参考）

语句关键字（C# 参考）

语句是程序指令。除了下表中引用的主题中介绍的以外，语句都是按照顺序执行的。下表列出了 C# 语句关键字。有关不用任何关键字表示的语句的更多信息，请参见语句（C# 编程指南）。

类别	C# 关键字
选择语句	if, else, switch, case
迭代语句	do, for, foreach, in, while
跳转语句	break, continue, default, goto, return, yield
异常处理语句	throw, try-catch, try-finally, try-catch-finally
检查和未检查	checked, unchecked
fixed 语句	fixed
lock 语句	lock

请参阅

C# 参考

语句（C# 编程指南）

C# 关键字

选择语句（C# 参考）

选择语句根据某个条件是否为 **true** 来将程序控制权移交给特定的流。

选择语句中使用下列关键字：

- if
- else
- switch
- case
- default

重要章节

Flow Control 在 Beginning Visual C# 2010

请参阅

C# 参考

C# 编程指南

C# 关键字

语句关键字（C# 参考）

if-else（C# 参考）

运行的语句根据 **Boolean** 表达式的值 **if** 语句标识。在下面的示例中，**Boolean** 变量 result 设置为 **true**，然后在 **if** 语句中检查该变量。输出为 The condition is true。

```
bool condition = true;

if (condition)
{
    Console.WriteLine("The variable is set to true.");
}
else
{
    Console.WriteLine("The variable is set to false.");
}
```

可以本主题中的示例通过将它们负责在控件个 app 的 **Main** 方法。

如下面的示例所示，在 c# 中为 **if** 语句可以采用两种形式。

```

// if-else statement
if (condition)
{
    then-statement;
}
else
{
    else-statement;
}
// Next statement in the program.

// if statement without an else
if (condition)
{
    then-statement;
}
// Next statement in the program.
```

在 **if-else** 语句，则为；condition 计算结果为 true，then-statement 运行。如果 condition 为 false，else-statement 运行。由于 condition 不能同时为 true 和 false，then-statement 和 **if-else** 语句的 else-statement 不能运行的两个。在 then-statement 或 else-statement 运行后，控件传输到下一条语句在 **if** 语句之后。

在没有包括一个 **else** 语句的 **if** 语句，则为；condition 为 true，then-statement 运行。如果 condition 为 false，控件传输到下一条语句在 **if** 语句之后。

在大括号的 then-statement 和 else-statement 可以包含一条或多条语句 (**{}**) 中。对于单个语句，大括号是可选的，但建议。

语句或语句。then-statement 和 else-statement 可以是任何类型，包括另一个 **if** 语句嵌套在原始 **if** 语句中。在嵌套 **if** 语句，没有相应的 **else**的每 **else** 子句属于最后 **if**。在下面的示例中，则为；m > 10 和 n > 20 计算结果为 true，Result1 显示。如果 m > 10 为 true，但 n > 20 为 false，Result2 显示。

```
// Try with m = 12 and then with m = 8.
int m = 12;
int n = 18;

if (m > 10)
    if (n > 20)
    {
        Console.WriteLine("Result1");
    }
    else
    {
        Console.WriteLine("Result2");
    }
```

如果为，则相反，您希望 Result2 显示 (m > 10) 为 false，您可以指定该关联。使用大括号建立嵌套 **if** 语句的开头和结尾，如下例所示。

```
// Try with m = 12 and then with m = 8.
if (m > 10)
{
    if (n > 20)
        Console.WriteLine("Result1");
}
else
{
    Console.WriteLine("Result2");
}
```

在条件 (m > 10) 计算为 FALSE，Result2 显示。

在下面的示例中，将从键盘输入字符，因此，程序使用嵌套的 **if** 语句来确定输入字符是否为字母字符。如果输入字符是一个字母，程序检查输入字符是否例或大写。消息为每个用例显示。

```
Console.Write("Enter a character: ");
char c = (char)Console.Read();
if (Char.IsLetter(c))
{
    if (Char.IsLower(c))
    {
        Console.WriteLine("The character is lowercase.");
    }
    else
    {
        Console.WriteLine("The character is uppercase.");
    }
}
else
{
    Console.WriteLine("The character isn't an alphabetic character
}

//Sample Output:

//Enter a character: 2
//The character isn't an alphabetic character.

//Enter a character: A
//The character is uppercase.

//Enter a character: h
//The character is lowercase.
```

如下面的代码部分显示，还可以嵌套在其他内的某个 **if** 语句块。该示例嵌套在两内的 **if** 语句其他块，并人块。注释指定的条件为 true 或 false 在每个块。

```
// Change the values of these variables to test the results.
bool Condition1 = true;
bool Condition2 = true;
bool Condition3 = true;
bool Condition4 = true;

if (Condition1)
{
    // Condition1 is true.
}
else if (Condition2)
{
    // Condition1 is false and Condition2 is true.
}
else if (Condition3)
{
    if (Condition4)
    {
        // Condition1 and Condition2 are false. Condition3 and Conc
    }
    else
    {
        // Condition1, Condition2, and Condition4 are false. Condit
    }
}
else
{
    // Condition1, Condition2, and Condition3 are false.
}
```

下面的示例确定输入的字符是否是一个小写字母、大写字母或数字。如果所有三个条件为 false，字符是字母数字字符。该示例演示每种情况的一条消息。

```
Console.Write("Enter a character: ");
char ch = (char)Console.Read();

if (Char.IsUpper(ch))
{
    Console.WriteLine("The character is an uppercase letter.");
}
else if (Char.IsLower(ch))
{
    Console.WriteLine("The character is a lowercase letter.");
}
else if (Char.IsDigit(ch))
{
    Console.WriteLine("The character is a number.");
}
else
{
    Console.WriteLine("The character is not alphanumeric.");
}

//Sample Input and Output:
//Enter a character: E
//The character is an uppercase letter.

//Enter a character: e
//The character is a lowercase letter.

//Enter a character: 4
//The character is a number.

//Enter a character: =
//The character is not alphanumeric.
```

正如在另一个语句块或块可以是任何有效的语句，可以为该条件使用任何有效的布尔值表示样式。您可以使用逻辑运算符 (例如 &&，&，||，|并使多个条件的 !。下面的代码演示示例。

```
// NOT
bool result = true;
if (!result)
{
    Console.WriteLine("The condition is true (result is false).");
}
else
{
    Console.WriteLine("The condition is false (result is true).");
}

// Short-circuit AND
int m = 9;
int n = 7;
int p = 5;
if (m >= n && m >= p)
{
    Console.WriteLine("Nothing is larger than m.");
}

// AND and NOT
if (m >= n && !(p > m))
{
    Console.WriteLine("Nothing is larger than m.");
}

// Short-circuit OR
if (m > n || m > p)
{
    Console.WriteLine("m isn't the smallest.");
}

// NOT and OR
m = 4;
if (!(m >= n || m >= p))
{
    Console.WriteLine("Now m is the smallest.");
}
// Output:
// The condition is false (result is true).
// Nothing is larger than m.
// Nothing is larger than m.
// m isn't the smallest.
// Now m is the smallest.
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

?: 运算符（C# 参考）

if-else 语句 (C++)

switch（C# 参考）

switch（C# 参考）

switch 语句是一个控制语句，用于从候选列表中选择一个要执行的开关部分。

switch 语句包含一个或多个开关部分。每个开关部分包含一个或多个 case 标签，后接一个或多个语句。下面的示例展示了一个包含三个开关部分的简单 **switch** 语句。每个开关部分各有一个 case 标签（例如 case 1）和两个语句。

```
int caseSwitch = 1;
switch (caseSwitch)
{
    case 1:
        Console.WriteLine("Case 1");
        break;
    case 2:
        Console.WriteLine("Case 2");
        break;
    default:
        Console.WriteLine("Default case");
        break;
}
```

备注

每个 case 标签指定一个常数值。switch 语句会将控制传输到 case 标签与 switch 表达式的值（示例中为 caseSwitch）相符的开关部分。如果任何 case 标签都不包含匹配值，则将控制传输到 **default** 部分（如果有）。如果没有 **default** 部分，则不会执行任何操作，并在 **switch** 语句之外传输控制。在上一个示例中，因为 case 1 与 caseSwitch 的值匹配，因此执行第一个开关部分中的语句。

switch 语句中可以包含任意数量的开关部分，每个开关部分可以具有一个或多个 case 标签（如下面的字符串 case 标签示例中所示）。但是，任何两个 case 标签不可包含相同的常数值。

执行选定开关部分中的语句列表时，将首先执行第一个语句，然后执行整个语句列表，通常直到到达一个跳转语句为止，如 **break**、**goto case**、**return** 或 **throw**。此时，控件在 **switch** 语句之外进行传输或传输到另一个 case 标签。

与 C++ 不同、C# 不允许从一个开关部分继续执行到下一个开关部分。下面的代码会导致错误。

```
switch (caseSwitch)
{
    // The following switch section causes an error.
    case 1:
        Console.WriteLine("Case 1...");
        // Add a break or other jump statement here.
    case 2:
        Console.WriteLine("... and/or Case 2");
        break;
}
```

C# 要求开关部分（包括最后一个）的末尾不可到达。就是说，不同于其他一些语言，代码不能落入下一个开关部分。虽然此要求通常使用 **break** 语句来满足，但以下情况同样有效，因为它可以确保无法到达语句列表的末尾。

```
case 4:
    while (true)
        Console.WriteLine("Endless looping. . . .");
```

下面的示例演示 **switch** 语句的要求和功能。

```
class Program
{
    static void Main(string[] args)
    {
        int switchExpression = 3;
        switch (switchExpression)
        {
            // A switch section can have more than one case label.
            case 0:
            case 1:
                Console.WriteLine("Case 0 or 1");
                // Most switch sections contain a jump statement, s
                // a break, goto, or return. The end of the stateme
                // must be unreachable.
                break;
            case 2:
                Console.WriteLine("Case 2");
                break;
                // The following line causes a warning.
                Console.WriteLine("Unreachable code");
            // 7 - 4 in the following line evaluates to 3.
            case 7 - 4:
                Console.WriteLine("Case 3");
                break;
            // If the value of switchExpression is not 0, 1, 2, or
            // default case is executed.
            default:
                Console.WriteLine("Default case (optional)");
                // You cannot "fall through" any switch section, in
                // the last one.
                break;
        }
    }
}
```

在最后一个示例中，字符串变量、str 和字符串 case 标签控制执行流。

```
class SwitchTest
{
    static void Main()
    {
        Console.WriteLine("Coffee sizes: 1=small 2=medium 3=large")
        Console.Write("Please enter your selection: ");
        string str = Console.ReadLine();
        int cost = 0;

        // Notice the goto statements in cases 2 and 3\. The base c
        // cents is added to the additional cost for the medium and
        switch (str)
        {
            case "1":
            case "small":
                cost += 25;
                break;
            case "2":
            case "medium":
                cost += 25;
                goto case "1";
            case "3":
            case "large":
                cost += 50;
                goto case "1";
            default:
                Console.WriteLine("Invalid selection. Please select
                break;
        }
        if (cost != 0)
        {
            Console.WriteLine("Please insert {0} cents.", cost);
        }
        Console.WriteLine("Thank you for your business.");
    }
}
/*
    Sample Input: 2

    Sample Output:
    Coffee sizes: 1=small 2=medium 3=large
    Please enter your selection: 2
    Please insert 50 cents.
    Thank you for your business.
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

switch 语句 (C++)

if-else（C# 参考）

迭代语句（C# 参考）

通过使用迭代语句可以创建循环。迭代语句导致嵌入语句根据循环终止条件多次执行。除非遇到跳转语句，否则这些语句将按顺序执行。

迭代语句中使用下列关键字：

- do
- for
- foreach
- in
- while

重要章节

Flow Control 在 Beginning Visual C# 2010

请参阅

C# 参考

C# 编程指南

C# 关键字

语句关键字（C# 参考）

do（C# 参考）

do 语句重复执行一个语句或语句块，直到指定的表达式计算为 **false** 值。循环体必须括在括号内，**{}**，除非它由单个语句组成。在这种情况下，大括号是可选的。

在下面的示例中，只要变量 x 小于 5，**do-while** 循环语句就开始执行。

```
public class TestDoWhile
{
    public static void Main ()
    {
        int x = 0;
        do
        {
            Console.WriteLine(x);
            x++;
        } while (x < 5);
    }
}
/*
    Output:
    0
    1
    2
    3
    4
*/
```

与 while 语句不同的是，**do-while** 循环会在计算条件表达式之前执行一次。

在 **do-while** 块中的任何点，都可使用 break 语句跳出循环。可通过使用 continue 语句直接步入 **while** 表达式评估语句。如果 **while** 表达式计算结果为 true，则继续执行循环中的第一个语句。如果表达式计算结果为 false，则会继续从 **do-while** 循环后的第一个语句执行。

do-while 循环还可以通过 goto、return 或 throw 语句退出。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

do-while 语句 (C++)

迭代语句（C# 参考）

for（C# 参考）

使用 **for** 循环，可以反复运行语句或语句块，直到指定的表达式计算为 **false**。这种循环是用于循环访问数组以及您事先知道的其他应用程序多少次希望此循环。

在下面的示例中，i 的值被写入控制台。并按 1 递增循环的每次迭代时。

```
class ForLoopTest
{
    static void Main()
    {
        for (int i = 1; i <= 5; i++)
        {
            Console.WriteLine(i);
        }
    }
}
/*
Output:
1
2
3
4
5
*/
```

前面示例中的 **for** 语句执行以下操作。

1. 首先，可变 i 的初始值建立的。此步骤仅发生一次，无论多少次此循环。您可以将此初始化，当发生循环处理的外部。
2. 若要计算该条件 (i <= 5)，i 的值与 5. 比较。
    - 如果 i 小于或等于 5，该条件的计算结果为 **true**，因此，以下操作。
        1. 在循环主体的 Console.WriteLine 语句演示 i的值。
        2. i 的值由 1. 增加。
        3. 循环回起点第 2 步再次计算该条件。
    - 如果 i 大于 5，该条件的计算结果为 **false**，因此，您退出循环。

请注意，因此，如果 i 的原始值大于 5，循环体哪怕一次不会运行。

每个 **for** 语句定义初始值设定项、条件和迭代器部分。这些部分通常多少次此循环。

```
for (initializer; condition; iterator)
    body
```

部分提供了以下用途。

- 初始值设定项部分设置初始条件。在任一次运行的此部分的语句，则，在进入循环之前。该部分只能包含下面两个选项。
  - 本地循环变量的声明和初始化，作为第一个示例显示 (int i = 1)。该变量是本地到循环，不能从循环外部访问。
  - 零个或多个语句 expressons 从以下列表，以逗号分隔。
    - 分配 语句
    - 方法的调用
    - 向或后缀 增量 表达式作为前缀，例如 ++i 或 i++
    - 向或后缀 减量 表达式作为前缀，例如 --i 或 i--
    - 对象的使用创建的 new
    - 等待 表达式
- 条件部分包含计算确定的布尔表达式循环是否应该退出或应再次运行。
- 迭代器节定义了什么在循环体中的每个迭代之后发生。迭代器节包含零个或多个下列语句表达式，逗号分隔):
  - 分配 语句
  - 方法的调用
  - 向或后缀 增量 表达式作为前缀，例如 ++i 或 i++
  - 向或后缀 减量 表达式作为前缀，例如 --i 或 i--
  - 对象的使用创建的 new
  - 等待 表达式
- 循环体包括语句、一个空的语句或语句块，则通过将零个或多个语句创建在大括号。

  使用 中断 关键字，则可能发生 **for** 循环，使用 继续 关键字，也可以单步执行到下一个迭代。使用 导航、返回或 引发 语句，还可以退出所有循环。

本主题中的第一个示例显示最典型的 **for** 循环，做出部分的下列选择。

- 该初始值设定项声明并初始化本地循环变量，i，维护循环的迭代计数。
- 状态检查循环变量的值已知的最终值，5. 的。

- 迭代器部分使用后缀递增语句，i++，匹配循环的每次迭代。

下面的示例阐释若干不太常见选择：将值赋给初始值设定项部分的外部循环变量，对初始值设定项和迭代器部分的 Console.WriteLine 方法和更改两个变量的值在迭代器部分。

```
static void Main()
 {
     int i;
     int j = 10;
     for (i = 0, Console.WriteLine("Start: {0}",i); i < j; i++, j--
     {
         // Body of the loop.
     }
 }
 // Output:
 // Start: 0
 // i=1, j=9
 // i=2, j=8
 // i=3, j=7
 // i=4, j=6
 // i=5, j=5
```

定义一个 **for** 语句的任何表达式都是可选的。例如，下面的语句创建无限循环。

```
for (; ; )
{
    // ...
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

foreach，in（C# 参考）

for 语句 (C++)

迭代语句（C# 参考）

foreach，in（C# 参考）

foreach 语句对实现 System.Collections.IEnumerable 或 System.Collections.Generic.IEnumerable<T> 接口的数组或对象集合中的每个元素重复一组嵌入式语句。 **foreach** 语句用于循环访问集合，以获取您需要的信息，但不能用于在源集合中添加或移除项，否则可能产生不可预知的副作用。如果需要在源集合中添加或移除项，请使用 for 循环。

嵌入语句为数组或集合中的每个元素继续执行。当为集合中的所有元素完成迭代后，控制传递给 **foreach** 块之后的下一个语句。

可以在 **foreach** 块的任何点使用 break 关键字跳出循环，或使用 continue 关键字进入循环的下一轮迭代。

foreach 循环还可以通过 goto、return 或 throw 语句退出。

有关 **foreach** 关键字和代码示例的更多信息，请参见下面的主题：

对数组使用 foreach（C# 编程指南）

如何：使用 foreach 访问集合类（C# 编程指南）

以下代码显示了三个示例：

- 显示整数数组内容的典型的 **foreach** 循环
- 执行相同操作的 for 循环
- 维护数组中的元素数计数的 **foreach** 循环

```
class ForEachTest
{
    static void Main(string[] args)
    {
        int[] fibarray = new int[] { 0, 1, 1, 2, 3, 5, 8, 13 };
        foreach (int element in fibarray)
        {
            System.Console.WriteLine(element);
        }
        System.Console.WriteLine();

        // Compare the previous loop to a similar for loop.
        for (int i = 0; i < fibarray.Length; i++)
        {
            System.Console.WriteLine(fibarray[i]);
        }
        System.Console.WriteLine();

        // You can maintain a count of the elements in the collect:
        int count = 0;
```

```
        foreach (int element in fibarray)
        {
            count += 1;
            System.Console.WriteLine("Element #{0}: {1}", count, el
        }
        System.Console.WriteLine("Number of elements in the array: 
    }
    // Output:
    // 0
    // 1
    // 1
    // 2
    // 3
    // 5
    // 8
    // 13

    // 0
    // 1
    // 1
    // 2
    // 3
    // 5
    // 8
    // 13

    // Element #1: 0
    // Element #2: 1
    // Element #3: 1
    // Element #4: 2
    // Element #5: 3
    // Element #6: 5
    // Element #7: 8
    // Element #8: 13
    // Number of elements in the array: 8
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

迭代语句（C# 参考）

for（C# 参考）

while（C# 参考）

while 语句执行一个语句或语句块，直到指定的表达式计算为 **false**。

```
class WhileTest
{
    static void Main()
    {
        int n = 1;
        while (n < 6)
        {
            Console.WriteLine("Current value of n is {0}", n);
            n++;
        }
    }
}
/*
    Output:
    Current value of n is 1
    Current value of n is 2
    Current value of n is 3
    Current value of n is 4
    Current value of n is 5
 */
```

```
class WhileTest2
{
    static void Main()
    {
        int n = 1;
        while (n++ < 6)
        {
            Console.WriteLine("Current value of n is {0}", n);
        }
    }
}
/*
Output:
Current value of n is 2
Current value of n is 3
Current value of n is 4
Current value of n is 5
Current value of n is 6
*/
```

由于 **while** 表达式的测试在每次执行循环前发生，因此 **while** 循环执行零次或更多次。这与执行一次或多次的 do 循环不同。

当 break、goto、return 或 throw 语句将控制权转移到 **while** 循环之外时，可以终止该循环。若要将控制权传递给下一次迭代但不退出循环，请使用 continue 语句。请注意，在上面三个示例中，根据 int n 递增的位置的不同，输出也不同。在下面的示例中不生成输出。

```
class WhileTest3
{
    static void Main()
    {
        int n = 5;
        while (++n < 6)
        {
            Console.WriteLine("Current value of n is {0}", n);
        }
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

While 语句 (C++)

迭代语句（C# 参考）

跳转语句（C# 参考）

使用跳转语句执行分支，该语句导致立即传递程序控制。跳转语句中使用下列关键字：

- break
- continue
- goto
- return
- throw

请参阅

C# 参考

C# 编程指南

C# 关键字

语句关键字（C# 参考）

break（C# 参考）

break 语句用于终止最近的封闭循环或它所在的 switch 语句。控制传递给终止语句后面的语句（如果有的话）。

在此示例中，条件语句包含一个应从 1 计数到 100 的计数器；但 **break** 语句在计数器计数到 4 后终止了循环。

```
class BreakTest
{
    static void Main()
    {
        for (int i = 1; i <= 100; i++)
        {
            if (i == 5)
            {
                break;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
 Output:
    1
    2
    3
    4
*/
```

在此示例中，**break** 语句用于中断内层嵌套循环，并将控制权返回给外层循环。

```
class BreakInNestedLoops
{
    static void Main(string[] args)
    {

        int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        char[] letters = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',

        // Outer loop
        for (int x = 0; x < numbers.Length; x++)
        {
            Console.WriteLine("num = {0}", numbers[x]);
```

```
            // Inner loop
            for (int y = 0; y < letters.Length; y++)
            {
                if (y == x)
                {
                    // Return control to outer loop
                    break;
                }
                Console.Write(" {0} ", letters[y]);
            }
            Console.WriteLine();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
 * Output:
    num = 0

    num = 1
     a
    num = 2
     a  b
    num = 3
     a  b  c
    num = 4
     a  b  c  d
    num = 5
     a  b  c  d  e
    num = 6
     a  b  c  d  e  f
    num = 7
     a  b  c  d  e  f  g
    num = 8
     a  b  c  d  e  f  g  h
    num = 9
     a  b  c  d  e  f  g  h  i
 */
```

下面的示例演示 **break** 在 switch 语句中的用法。

```
class Switch
{
    static void Main()
    {
        Console.Write("Enter your selection (1, 2, or 3): ");
        string s = Console.ReadLine();
        int n = Int32.Parse(s);

        switch (n)
        {
            case 1:
                Console.WriteLine("Current value is {0}", 1);
                break;
            case 2:
                Console.WriteLine("Current value is {0}", 2);
                break;
            case 3:
                Console.WriteLine("Current value is {0}", 3);
                break;
            default:
                Console.WriteLine("Sorry, invalid selection.");
                break;
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input: 1

Sample Output:
Enter your selection (1, 2, or 3): 1
Current value is 1
*/
```

如果输入了 4，则输出为：

```
Enter your selection (1, 2, or 3): 4
Sorry, invalid selection.
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

switch（C# 参考）

跳转语句（C# 参考）

迭代语句（C# 参考）

continue（C# 参考）

continue 语句将控制传递到封闭 while、、为或显示的 foreach 语句下一次迭代。

在本示例中，计数器最初是从 1 到 10 进行计数。通过使用该表达式 (i < 9)一起的 **continue** 语句，在 **continue** 和 **for** 主体结束的语句跳过。

```
class ContinueTest
{
    static void Main()
    {
        for (int i = 1; i <= 10; i++)
        {
            if (i < 9)
            {
                continue;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Output:
9
10
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

break 语句 (C++)

跳转语句（C# 参考）

goto（C# 参考）

goto 语句将程序控制直接传递给标记语句。

goto 的一个通常用法是将控制传递给特定的 switch-case 标签或 **switch** 语句中的默认标签。

goto 语句还用于跳出深嵌套循环。

下面的示例演示了 **goto** 在 switch 语句中的使用。

```
class SwitchTest
{
    static void Main()
    {
        Console.WriteLine("Coffee sizes: 1=Small 2=Medium 3=Large");
        Console.Write("Please enter your selection: ");
        string s = Console.ReadLine();
        int n = int.Parse(s);
        int cost = 0;
        switch (n)
        {
            case 1:
                cost += 25;
                break;
            case 2:
                cost += 25;
                goto case 1;
            case 3:
                cost += 50;
                goto case 1;
            default:
                Console.WriteLine("Invalid selection.");
                break;
        }
        if (cost != 0)
        {
            Console.WriteLine("Please insert {0} cents.", cost);
        }
        Console.WriteLine("Thank you for your business.");

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input:  2

Sample Output:
Coffee sizes: 1=Small 2=Medium 3=Large
Please enter your selection: 2
Please insert 50 cents.
Thank you for your business.
*/
```

下面的示例演示了使用 **goto** 跳出嵌套循环。

```
public class GotoTest1
{
    static void Main()
```

```
    {
        int x = 200, y = 4;
        int count = 0;
        string[,] array = new string[x, y];

        // Initialize the array:
        for (int i = 0; i < x; i++)

            for (int j = 0; j < y; j++)
                array[i, j] = (++count).ToString();

        // Read input:
        Console.Write("Enter the number to search for: ");

        // Input a string:
        string myNumber = Console.ReadLine();

        // Search:
        for (int i = 0; i < x; i++)
        {
            for (int j = 0; j < y; j++)
            {
                if (array[i, j].Equals(myNumber))
                {
                    goto Found;
                }
            }
        }

        Console.WriteLine("The number {0} was not found.", myNumber
        goto Finish;

    Found:
        Console.WriteLine("The number {0} is found.", myNumber);

    Finish:
        Console.WriteLine("End of search.");

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input: 44

Sample Output
Enter the number to search for: 44
The number 44 is found.
End of search.
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

goto 语句 (C++)

跳转语句（C# 参考）

return（C# 参考）

return 语句终止它出现在其中的方法的执行并将控制返回给调用方法。它还可以返回一个可选值。如果方法为 **void** 类型，则可以省略 **return** 语句。

如果 return 语句位于 **try** 块中，则将在控制流返回到调用方法之前执行 **finally** 块（如果存在）。

在下面的示例中，方法 A() 以 double 值的形式返回变量 Area。

```
class ReturnTest
{
    static double CalculateArea(int r)
    {
        double area = r * r * Math.PI;
        return area;
    }

    static void Main()
    {
        int radius = 5;
        double result = CalculateArea(radius);
        Console.WriteLine("The area is {0:0.00}", result);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: The area is 78.54
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

return 语句 (C++)

跳转语句（C# 参考）

异常处理语句（C# 参考）

C# 为处理在程序执行期间可能出现的反常情况（称作异常）提供内置支持。这些异常由正常控制流之外的代码处理。

本节说明下列异常处理主题：

- throw
- try-catch
- try-finally
- try-catch-finally

请参阅

C# 参考

C# 编程指南

C# 关键字

语句关键字（C# 参考）

异常和异常处理（C# 编程指南）

throw（C# 参考）

throw 语句用于发出程序执行期间出现反常情况（异常）的信号。

备注

引发的异常是一个对象，其类派生自 System.Exception，如以下示例所示。

```
class MyException : System.Exception {}
// ...
throw new MyException();
```

通常，**throw** 语句与 **try-catch** 或 **try-finally** 语句结合使用。可在 **catch** 块中使用语句以重新引发已由 **catch** 块捕获的异常。在这种情况下，语句不采用异常操作数。有关更多信息和示例，请参见 try-catch（C# 参考）和如何：显式引发异常。

此示例演示如何使用 **throw** 语句引发异常。

```
public class ThrowTest2
{

    static int GetNumber(int index)
    {
        int[] nums = { 300, 600, 900 };
        if (index > nums.Length)
        {
            throw new IndexOutOfRangeException();
        }
        return nums[index];

    }
    static void Main()
    {
        int result = GetNumber(3);

    }
}
/*
    Output:
    The System.IndexOutOfRangeException exception occurs.
*/
```

代码示例

请参见 try-catch（C# 参考）和如何：显式引发异常中的示例。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

try-catch（C# 参考）

C++ 中的 try、catch 和 throw 语句

C# 关键字

异常处理语句（C# 参考）

如何：显式引发异常

try-catch（C# 参考）

Try-catch 语句包含一个后接一个或多个 **catch** 子句的 **try** 块，这些子句指定不同异常的处理程序。

备注

引发异常时，公共语言运行时 (CLR) 查找处理此异常的 **catch** 语句。如果当前正在执行的方法不包含此类 **catch** 块，则 CLR 查看调用了当前方法的方法，并以此类推遍历调用堆栈。如果未找到任何 **catch** 块，则 CLR 向用户显示一条未处理的异常消息，并停止执行程序。

try 块包含可能导致异常的受保护的代码。将执行此块，直至引发异常或其成功完成。例如，强制转换 **null** 对象的以下尝试会引发 NullReferenceException 异常：

```
object o2 = null;
try
{
    int i2 = (int)o2;   // Error
}
```

尽管可以不带参数使用 **catch** 子句来捕获任何类型的异常，但不推荐这种用法。一般情况下，只应捕获你知道如何从其恢复的异常。因此，应始终指定派生自 System.Exception 的对象参数，例如：

```
catch (InvalidCastException e)
{
}
```

可以使用同一 try-catch 语句中的多个特定 **catch** 子句。在这种情况下，**catch** 子句的顺序很重要，因为 **catch** 子句是按顺序检查的。在使用更笼统的子句之前获取特定性更强的异常。如果捕获块的排序使得永不会达到之后的块，则编译器将产生错误。

筛选想要处理的异常的一种方式是使用 **catch** 参数。也可以使用谓词表达式进一步检查该异常以决定是否要对其进行处理。如果谓词表达式返回 false，则继续搜索处理程序。

```
Catch (ArgumentException e) if (e.ParamName == "…")
{
}
```

异常筛选器要优于捕获和重新引发（如下所述），因为筛选器将保留堆栈不受损坏。如果之后的处理程序转储堆栈，可以查看到异常的原始来源，而不只是重新引发它的最后一个位置。异常筛选器表达式的一个常见用途是日志记录。可以创建一个始终返回 false 并输出到日志的谓词函数，而且可以在异常通过时进行记录，无需处理并重新引发它们。

可在 **catch** 块中使用 throw 语句以重新引发已由 **catch** 语句捕获的异常。下面的示例从 IOException 异常提取源信息，然后向父方法引发异常。

```
catch (FileNotFoundException e)
{
    // FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
    // Extract some information from this exception, and then
    // throw it to the parent method.
    if (e.Source != null)
        Console.WriteLine("IOException source: {0}", e.Source);
    throw;
}
```

你可以捕获一个异常而引发一个不同的异常。执行此操作时，请指定作为内部异常捕获的异常，如以下示例所示。

```
catch (InvalidCastException e)
{
    // Perform some action here, and then throw a new exception.
    throw new YourCustomException("Put your error message here.", e
}
```

当指定的条件为 true 时，你还可以重新引发异常，如以下示例所示。

```

catch (InvalidCastException e)
{
    if (e.Data == null)
    {
        throw;
    }
    else
    {
        // Take some action.
    }
 }
```

从 **try** 块内，仅初始化在其中声明的变量。否则，在完成执行块之前，可能会出现异常。例如，在下面的代码示例中，变量 n 在 **try** 块内部初始化。尝试在 Write(n) 语句的 **try** 块外部使用此变量将生成编译器错误。

```
static void Main()
{
    int n;
    try
    {
        // Do not initialize this variable here.
        n = 123;
    }
    catch
    {
    }
    // Error: Use of unassigned local variable 'n'.
    Console.Write(n);
}
```

有关 catch 的详细信息，请参阅 try-catch-finally。

异步方法中的异常

异步方法由 async 修饰符标记，通常包含一个或多个 await 表达式或语句。await 表达式将 await 运算符应用于 Task 或 Task<TResult>。

当控件到达异步方法中的 **await** 时，将挂起方法中的进度，直到所等待的任务完成。任务完成后，可以在方法中恢复执行。有关详细信息，请参阅使用 Async 和 Await 的异步编程（C# 和 Visual Basic）和异步程序中的控制流（C# 和 Visual Basic）。

应用了 **await** 的完成任务可能由于返回此任务的方法中存在未处理的异常而处于错误状态。等待该任务引发异常。如果取消了返回任务的异步进程，此任务最后也可能为已取消状态。等待已取消的任务引发 **OperationCanceledException**。有关如何取消异步进程的详细信息，请参阅 微调异步应用程序（C# 和 Visual Basic）。

若要捕获异常，请在 **try** 块中等待任务并在关联的 **catch** 块中捕获异常。相关示例，请参见“示例”一节。

任务可能处于错误状态，因为等待的异步方法中发生了多个异常。例如，任务可能是对 Task.WhenAll 调用的结果。当等待此类任务时，仅捕捉到其中一个异常，而且你无法预测将会捕获到哪个异常。相关示例，请参见“示例”一节。

在下面的示例中，**try** 块包含对可能引发异常的 ProcessString 方法的调用。 **catch** 子句包含只在屏幕上显示一条消息的异常处理程序。当从 MyMethod 内部调用 **throw** 语句时，系统将查找 **catch** 语句并显示消息 Exception caught。

```
    class TryFinallyTest
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    static void Main()
    {
        string s = null; // For demonstration purposes.

        try
        {
            ProcessString(s);
        }

        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}
    /*
    Output:
    System.ArgumentNullException: Value cannot be null.
       at TryFinallyTest.Main() Exception caught.
     * */
```

在下面的示例中，使用了两个 catch 块，并捕获到最先出现的最具体的异常。

若要捕获最不具体的异常，你可以将 ProcessString 中的 throw 语句替换为以下语句：throw new Exception()。

如果将最不具体的 catch 块置于示例中第一个，将显示以下错误消息：A previous catch clause already catches all exceptions of this or a super type ('System.Exception')。

```
class ThrowTest3
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    static void Main()
    {
        try
        {
            string s = null;
            ProcessString(s);
        }
        // Most specific:
        catch (ArgumentNullException e)
        {
            Console.WriteLine("{0} First exception caught.", e);
        }
        // Least specific:
        catch (Exception e)
        {
            Console.WriteLine("{0} Second exception caught.", e);
        }
    }
}
/*
 Output:
 System.ArgumentNullException: Value cannot be null.
 at Test.ThrowTest3.ProcessString(String s) ... First exception cau
*/
```

下面的示例阐释异步方法的异常处理。若要捕获异步任务引发的异常，将 **await** 表达式置于 **try** 块中，并在 **catch** 块中捕获该异常。

将演示异常处理的示例中的 Throw New Exception 行取消注释。 任务的 **IsFaulted** 属性设置为 **True**，任务的 **Exception.InnerException** 属性设置为异常，并在 **catch** 块中捕获该异常。

取消注释 throw new OperationCancelledException 行以演示在取消异步进程时发生的情况。任务的 **IsCanceled** 属性设置为 **true**，并在 **catch** 块中捕获异常。在某些不适用于此示例的情况下，任务的 **IsFaulted** 属性设置为 **true** 且 **IsCanceled** 设置为 **false**。

```
public async Task DoSomethingAsync()
{
```

```
    Task<string> theTask = DelayAsync();

    try
    {
        string result = await theTask;
        Debug.WriteLine("Result: " + result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception Message: " + ex.Message);
    }
    Debug.WriteLine("Task IsCanceled: " + theTask.IsCanceled);
    Debug.WriteLine("Task IsFaulted:  " + theTask.IsFaulted);
    if (theTask.Exception != null)
    {
        Debug.WriteLine("Task Exception Message: "
            + theTask.Exception.Message);
        Debug.WriteLine("Task Inner Exception Message: "
            + theTask.Exception.InnerException.Message);
    }
}

private async Task<string> DelayAsync()
{
    await Task.Delay(100);

    // Uncomment each of the following lines to
    // demonstrate exception handling.

    //throw new OperationCanceledException("canceled");
    //throw new Exception("Something happened.");
    return "Done";
}

// Output when no exception is thrown in the awaited method:
//   Result: Done
//   Task IsCanceled: False
//   Task IsFaulted:  False

// Output when an Exception is thrown in the awaited method:
//   Exception Message: Something happened.
//   Task IsCanceled: False
//   Task IsFaulted:  True
//   Task Exception Message: One or more errors occurred.
//   Task Inner Exception Message: Something happened.

// Output when a OperationCanceledException or TaskCanceledExceptic
// is thrown in the awaited method:
//   Exception Message: canceled
//   Task IsCanceled: True
//   Task IsFaulted:  False
```

下面的示例阐释了在多个任务可能导致多个异常的情况中的异常处理。 **try** 块等待由 Task.WhenAll 的调用返回的任务。应用了 WhenAll 的三个任务完成后，该任务完成。

三个任务中的每一个都会导致异常。 **catch** 块循环访问异常，这些异常位于由 Task.WhenAll 返回的任务的 **Exception.InnerExceptions** 属性中。

```
public async Task DoMultipleAsync()
{
    Task theTask1 = ExcAsync(info: "First Task");
    Task theTask2 = ExcAsync(info: "Second Task");
    Task theTask3 = ExcAsync(info: "Third Task");

    Task allTasks = Task.WhenAll(theTask1, theTask2, theTask3);

    try
    {
        await allTasks;
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception: " + ex.Message);
        Debug.WriteLine("Task IsFaulted: " + allTasks.IsFaulted);
        foreach (var inEx in allTasks.Exception.InnerExceptions)
        {
            Debug.WriteLine("Task Inner Exception: " + inEx.Message
        }
    }
}

private async Task ExcAsync(string info)
{
    await Task.Delay(100);

    throw new Exception("Error-" + info);
}

// Output:
//   Exception: Error-First Task
//   Task IsFaulted: True
//   Task Inner Exception: Error-First Task
//   Task Inner Exception: Error-Second Task
//   Task Inner Exception: Error-Third Task
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

try、throw 和 catch 语句 (C++)

异常处理语句（C# 参考）

throw（C# 参考）

try-finally（C# 参考）

如何：显式引发异常

try-finally（C# 参考）

使用 **finally** 块，可以清理在 Try 中分配的任何资源，而且，即使在 **try** 块中发生异常，您也可以运行代码。 通常，控件离开 **try** 语句之后，**finally** 的语句会阻止运行。 正常执行中 **break**、**continue**、**goto** 或 **return** 语句的执行，或对 **try** 语句外部异常的传播，可能会导致发生控件转换。

已处理的异常中会确保运行关联的 **finally** 块。 但是，如果异常未得到处理，则 **finally** 块的执行取决于如何触发异常展开操作。 此操作又取决于计算机是如何设置的。 有关更多信息，请参见 Unhandled Exception Processing in the CLR（CLR 中的未经处理的异常处理）。

通常，当未经处理的异常中止应用程序时，**finally** 块是否运行并不重要。 但是，如果您拥有的 **finally** 块中的语句必须在该环境下运行，则一个解决方案是将 **catch** 块添加到 **try-finally** 语句中。 或者，可以捕获可能是在调用堆栈更上方的 **try-finally** 语句的 **try** 块中引发的异常。 即可以捕获调用了包含 **try-finally** 语句的方法中的、或调用了该方法的方法中的、或调用堆栈中任何方法中的异常。 如果未捕获异常，则 **finally** 块的执行取决于操作系统是否选择触发异常展开操作。

示例

在下面的示例中，无效转换语句导致 System.InvalidCastException 异常。 异常未处理。

```
public class ThrowTestA
{
    static void Main()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a nur
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try bloc
        }
        finally
        {
            // To run the program in Visual Studio, type CTRL+F5\.
            // click Cancel in the error dialog.
            Console.WriteLine("\nExecution of the finally block aft
                "error depends on how the exception unwind operatic
            Console.WriteLine("i = {0}", i);
        }
    }
    // Output:
    // Unhandled Exception: System.InvalidCastException: Specified
    //
    // Execution of the finally block after an unhandled
    // error depends on how the exception unwind operation is trigg
    // i = 123
}
```

在下面的示例中，TryCast 方法中的异常在延伸调用堆栈的方法中被捕获。

```
public class ThrowTestB
{
    static void Main()
    {
        try
        {
            // TryCast produces an unhandled exception.
            TryCast();
        }
        catch (Exception ex)
        {
            // Catch the exception that is unhandled in TryCast.
            Console.WriteLine
                ("Catching the {0} exception triggers the finally k
                ex.GetType());
```

```
                // Restore the original unhandled exception. You might
                // know what exception to expect, or how to handle it,
                // it on.
                throw;
            }
        }

        public static void TryCast()
        {
            int i = 123;
            string s = "Some string";
            object obj = s;

            try
            {
                // Invalid conversion; obj contains a string, not a nur
                i = (int)obj;

                // The following statement is not run.
                Console.WriteLine("WriteLine at the end of the try bloc
            }
            finally
            {
                // Report that the finally block is run, and show that
                // i has not been changed.
                Console.WriteLine("\nIn the finally block in TryCast, i
            }
        }
        // Output:
        // In the finally block in TryCast, i = 123.

        // Catching the System.InvalidCastException exception triggers

        // Unhandled Exception: System.InvalidCastException: Specified
    }
```

有关 **finally** 的更多信息，请参见 try-catch-finally。

C# 还包含使用语句，该语句可为 IDisposable 对象提供类似功能的方法语法。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

try、throw 和 catch 语句 (C++)

异常处理语句（C# 参考）

throw（C# 参考）

try-catch（C# 参考）

如何：显式引发异常

try-catch-finally（C# 参考）

catch 和 **finally** 一起使用的常见方式是：在 **try** 块中获取并使用资源，在 **catch** 块中处理异常情况，并在 **finally** 块中释放资源。

有关重新引发异常的更多信息和示例，请参见 try-catch 和“引发异常”。有关 **finally** 的更多信息，请参见 尝试最终块。

```
public class EHClass
{
    void ReadFile(int index)
    {
        // To run this code, substitute a valid path from your loca
        string path = @"c:\users\public\test.txt";
        System.IO.StreamReader file = new System.IO.StreamReader(pa
        char[] buffer = new char[10];
        try
        {
            file.ReadBlock(buffer, index, buffer.Length);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine("Error reading from {0}. Message = {1
        }

        finally
        {
            if (file != null)
            {
                file.Close();
            }
        }
        // Do something with buffer...
    }

}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

try、throw 和 catch 语句 (C++)

异常处理语句（C# 参考）

throw（C# 参考）

如何：显式引发异常

using 语句（C# 参考）

Checked 和 Unchecked（C# 参考）

C# 语句既可以在已检查的上下文中执行，也可以在未检查的上下文中执行。在已检查的上下文中，算法溢出引发异常。在未检查的上下文中，算法溢出被忽略并且结果被截断。

- checked 指定已检查的上下文。
- unchecked 指定未检查的上下文。

如果既未指定 **checked** 也未指定 **unchecked**，则默认上下文取决于外部因素（如编译器选项）。

下列操作受溢出检查的影响：

- 表达式在整型上使用下列预定义运算符：

  ++ -- -（一元） **+ - */***

- 整型间的显式数字转换。

/checked 编译器选项使你可以为 **checked** 或 **unchecked** 关键字范围内的所有非显式整型算术语句指定已检查或未检查的上下文。

请参阅

C# 参考

C# 编程指南

C# 关键字

语句关键字（C# 参考）

checked（C# 参考）

checked 关键字用于对整型算术运算和转换显式启用溢出检查。

默认情况下，如果表达式仅包含常数值，且产生的值在目标类型范围之外，则它会导致编译器错误。如果表达式包含一个或多个非常数值，则编译器不检测溢出。在下面的示例中，计算赋给 i2 的表达式不会导致编译器错误。

```
// The following example causes compiler error CS0220 because 2147
// is the maximum value for integers.
//int i1 = 2147483647 + 10;

// The following example, which includes variable ten, does not cau
// a compiler error.
int ten = 10;
int i2 = 2147483647 + ten;

// By default, the overflow in the previous statement also does
// not cause a run-time exception. The following line displays
// -2,147,483,639 as the sum of 2,147,483,647 and 10.
Console.WriteLine(i2);
```

默认情况下，在运行时也不检查这些非常数表达式是否溢出，这些表达式不引发溢出异常。上面的示例显示 -2,147,483,639 作为两个正整数之和。

可以通过编译器选项、环境配置或使用 **checked** 关键字来启用溢出检查。下面的示例演示如何使用 **checked** 表达式或 **checked** 块，在运行时检测由前面的求和计算导致的溢出。两个示例都引发溢出异常。

```
// If the previous sum is attempted in a checked environment, an
// OverflowException error is raised.

// Checked expression.
Console.WriteLine(checked(2147483647 + ten));

// Checked block.
checked
{
    int i3 = 2147483647 + ten;
    Console.WriteLine(i3);
}
```

可以使用 unchecked 关键字阻止溢出检查。

此示例演示如何使用 **checked** 启用运行时溢出检查。

```
class OverFlowTest
{
    // Set maxIntValue to the maximum value for integers.
    static int maxIntValue = 2147483647;

    // Using a checked expression.
    static int CheckedMethod()
    {
        int z = 0;
        try
        {
            // The following line raises an exception because it is
            z = checked(maxIntValue + 10);
        }
        catch (System.OverflowException e)
        {
            // The following line displays information about the er
            Console.WriteLine("CHECKED and CAUGHT:  " + e.ToString(
        }
        // The value of z is still 0.
        return z;
    }

    // Using an unchecked expression.
    static int UncheckedMethod()
    {
        int z = 0;
        try
        {
            // The following calculation is unchecked and will not
            // raise an exception.
            z = maxIntValue + 10;
        }
        catch (System.OverflowException e)
        {
            // The following line will not be executed.
            Console.WriteLine("UNCHECKED and CAUGHT:  " + e.ToStrin
        }
        // Because of the undetected overflow, the sum of 214748364
        // returned as -2147483639.
        return z;
    }

    static void Main()
    {
        Console.WriteLine("\nCHECKED output value is: {0}",
                          CheckedMethod());
        Console.WriteLine("UNCHECKED output value is: {0}",
                          UncheckedMethod());
    }
    /*
   Output:
```

```
    CHECKED and CAUGHT:  System.OverflowException: Arithmetic operat
    in an overflow.
       at ConsoleApplication1.OverFlowTest.CheckedMethod()

    CHECKED output value is: 0
    UNCHECKED output value is: -2147483639
  */
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

Checked 和 Unchecked（C# 参考）

unchecked（C# 参考）

unchecked（C# 参考）

unchecked 关键字用于取消整型算术运算和转换的溢出检查。

在未检查的上下文中，如果表达式产生的值在目标类型范围之外，并不会标记溢出。例如，下例中的计算在 **unchecked** 块或表达式中执行，因此将忽略结果对于整数而言过大这一事实，并会对 int1 赋予值 -2,147,483,639。

```
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

如果移除 **unchecked** 环境，则发生编译错误。因为表达式的各个项都是常数，所以可以在编译时检测到溢出。

默认情况下，在编译时和运行时不检查包含非常数项的表达式。有关启用检查环境的信息，请参见 checked（C# 参考）。

因为溢出检查比较耗时，所以当无溢出危险时，使用不检查的代码可以提高性能。但是，如果可能发生溢出，则应使用检查环境。

此示例演示如何使用 **unchecked** 关键字。

```
class UncheckedDemo
{
    static void Main(string[] args)
    {
        // int.MaxValue is 2,147,483,647.
        const int ConstantMax = int.MaxValue;
        int int1;
        int int2;
        int variableMax = 2147483647;

        // The following statements are checked by default at comp:
        // compile.
        //int1 = 2147483647 + 10;
        //int1 = ConstantMax + 10;

        // To enable the assignments to int1 to compile and run, p:
        // an unchecked block or expression. The following statemer
        // run.
        unchecked
        {
            int1 = 2147483647 + 10;
        }
        int1 = unchecked(ConstantMax + 10);
```

```
            // The sum of 2,147,483,647 and 10 is displayed as -2,147,4
            Console.WriteLine(int1);

            // The following statement is unchecked by default at compi
            // time because the expression contains the variable variab
            // overflow but the overflow is not detected. The statement
            int2 = variableMax + 10;

            // Again, the sum of 2,147,483,647 and 10 is displayed as -
            Console.WriteLine(int2);

            // To catch the overflow in the assignment to int2 at run t
            // declaration in a checked block or expression. The follow
            // statements compile but raise an overflow exception at ru
            checked
            {
                //int2 = variableMax + 10;
            }
            //int2 = checked(variableMax + 10);

            // Unchecked sections frequently are used to break out of a
            // environment in order to improve performance in a portion
            // that is not expected to raise overflow exceptions.
            checked
            {
                // Code that might cause overflow should be executed in
                // environment.
                unchecked
                {
                    // This section is appropriate for code that you ar
                    // will not result in overflow, and for which perfo
                    // a priority.
                }
                // Additional checked code here.
            }
        }
    }
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

Checked 和 Unchecked（C# 参考）

checked（C# 参考）

fixed 语句（C# 参考）

fixed 语句禁止垃圾回收器重定位可移动的变量。 **fixed** 语句只在不安全的上下文中是允许的。 **Fixed** 还可用于创建固定大小缓冲区。

fixed 语句设置指向托管变量的指针，并在执行该语句期间“固定”此变量。如果没有 **fixed** 语句，则指向可移动托管变量的指针的作用很小，因为垃圾回收可能不可预知地重定位变量。C# 编译器只允许在 **fixed** 语句中分配指向托管变量的指针。

```
unsafe static void TestMethod()
{

    // Assume that the following class exists.
    //class Point
    //{
    //    public int x;
    //    public int y;
    //}

    // Variable pt is a managed variable, subject to garbage collec
    Point pt = new Point();

    // Using fixed allows the address of pt members to be taken,
    // and "pins" pt so that it is not relocated.

    fixed (int* p = &pt.x)
    {
        *p = 1;
    }

}
```

可通过使用数组、字符串、大小固定的缓冲区或变量地址初始化指针。以下示例演示可变地址、数组和字符串的用法。关于固定大小缓冲器的更多信息，请参见 固定大小的缓冲区（C# 编程指南）。

```
static unsafe void Test2()
{
    Point point = new Point();
    double[] arr = { 0, 1.5, 2.3, 3.4, 4.0, 5.9 };
    string str = "Hello World";

    // The following two assignments are equivalent. Each assigns t
    // of the first element in array arr to pointer p.

    // You can initialize a pointer by using an array.
    fixed (double* p = arr) { /*...*/ }

    // You can initialize a pointer by using the address of a varia
    fixed (double* p = &arr[0]) { /*...*/ }

    // The following assignment initializes p by using a string.
    fixed (char* p = str) { /*...*/ }

    // The following assignment is not valid, because str[0] is a c
    // which is a value, not a variable.
    //fixed (char* p = &str[0]) { /*...*/ }

    // You can initialize a pointer by using the address of a varia
    // as point.x or arr[5].
    fixed (int* p1 = &point.x)
    {
        fixed (double* p2 = &arr[5])
        {
            // Do something with p1 and p2.
        }
    }
}
```

只要指针的类型相同，就可以初始化多个指针。

```
fixed (byte* ps = srcarray, pd = dstarray) {...}
```

要初始化不同类型的指针，只需嵌套 **fixed** 语句，如下面的示例所示。

```
fixed (int* p1 = &point.x)
{
    fixed (double* p2 = &arr[5])
    {
        // Do something with p1 and p2.
    }
}
```

执行完语句中的代码后，任何固定变量都被解除固定并受垃圾回收的制约。因此，不要指向 **fixed** 语句之外的那些变量。

注意
无法修改在 fixed 语句中初始化的指针。

在不安全模式中，可以在堆栈上分配内存。堆栈不受垃圾回收的制约，因此不需要被锁定。有关更多信息，请参见 stackalloc。

```
class Point
{
    public int x, y;
}

class FixedTest2
{
    // Unsafe method: takes a pointer to an int.
    unsafe static void SquarePtrParam (int* p)
    {
        *p *= *p;
    }

    unsafe static void Main()
    {
        Point pt = new Point();
        pt.x = 5;
        pt.y = 6;
        // Pin pt in place:
        fixed (int* p = &pt.x)
        {
            SquarePtrParam (p);
        }
        // pt now unpinned.
        Console.WriteLine ("{0} {1}", pt.x, pt.y);
    }
}
/*
Output:
25 6
 */
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

unsafe（C# 参考）

固定大小的缓冲区（C# 编程指南）

“锁定”语句（C# 参考）

lock 关键字将语句块标记为临界区，方法是获取给定对象的互斥锁，执行语句，然后释放该锁。下面的示例包含一个 **lock** 语句。

```
class Account
{
    decimal balance;
    private Object thisLock = new Object();

    public void Withdraw(decimal amount)
    {
        lock (thisLock)
        {
            if (amount > balance)
            {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}
```

有关更多信息，请参见 线程同步（C# 和 Visual Basic）。

备注

lock 关键字可确保当一个线程位于代码的临界区时，另一个线程不会进入该临界区。如果其他线程尝试进入锁定的代码，则它将一直等待（即被阻止），直到该对象被释放。

线程处理（C# 和 Visual Basic） 这节讨论了线程处理。

lock 关键字在块的开始处调用 Enter，而在块的结尾处调用 Exit。ThreadInterruptedException 引发，如果 Interrupt 中断等待输入 **lock** 语句的线程。

通常，应避免锁定 **public** 类型，否则实例将超出代码的控制范围。常见的结构 lock (this)、lock (typeof (MyType)) 和 lock ("myLock") 违反此准则：

- 如果实例可以被公共访问，将出现 lock (this) 问题。
- 如果 MyType 可以被公共访问，将出现 lock (typeof (MyType)) 问题。
- 由于进程中使用同一字符串的任何其他代码都将共享同一个锁，所以出现 lock("myLock") 问题。

最佳做法是定义 **private** 对象来锁定, 或 **private static** 对象变量来保护所有实例所共有的数据。

在 **lock** 语句的正文不能使用 等待 关键字。

下面演示在 C# 中使用未锁定的线程的简单示例。

```
//using System.Threading;

class ThreadTest
{
    public void RunMe()
    {
        Console.WriteLine("RunMe called");
    }

    static void Main()
    {
        ThreadTest b = new ThreadTest();
        Thread t = new Thread(b.RunMe);
        t.Start();
    }
}
// Output: RunMe called
```

下例使用线程和 **lock**。只要 **lock** 语句存在，语句块就是临界区并且 balance 永远不会是负数。

```
// using System.Threading;

class Account
{
    private Object thisLock = new Object();
    int balance;

    Random r = new Random();

    public Account(int initial)
    {
        balance = initial;
    }

    int Withdraw(int amount)
    {

        // This condition never is true unless the lock statement
        // is commented out.
        if (balance < 0)
        {
            throw new Exception("Negative Balance");
        }
```

```
            // Comment out the next line to see the effect of leaving o
            // the lock keyword.
            lock (thisLock)
            {
                if (balance >= amount)
                {
                    Console.WriteLine("Balance before Withdrawal :  " -
                    Console.WriteLine("Amount to Withdraw        : -" -
                    balance = balance - amount;
                    Console.WriteLine("Balance after Withdrawal  :  " -
                    return amount;
                }
                else
                {
                    return 0; // transaction rejected
                }
            }
        }

        public void DoTransactions()
        {
            for (int i = 0; i < 100; i++)
            {
                Withdraw(r.Next(1, 100));
            }
        }
    }

    class Test
    {
        static void Main()
        {
            Thread[] threads = new Thread[10];
            Account acc = new Account(1000);
            for (int i = 0; i < 10; i++)
            {
                Thread t = new Thread(new ThreadStart(acc.DoTransactior
                threads[i] = t;
            }
            for (int i = 0; i < 10; i++)
            {
                threads[i].Start();
            }
        }
    }
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

MethodImplAttributes

Mutex

C# 参考

C# 编程指南

线程处理（C# 和 Visual Basic）

C# 关键字

语句关键字（C# 参考）

监视器

Interlocked Operations

AutoResetEvent

线程同步（C# 和 Visual Basic）

方法参数（C# 参考）

如果在为方法声明参数时未使用 ref 或 out，则该参数可以具有关联的值。可以在方法中更改该值，但当控制传递回调用过程时，不会保留更改的值。通过使用方法参数关键字，可以更改这种行为。

本节描述声明方法参数时可以使用的关键字：

- params
- ref
- out

请参阅

C# 参考

C# 编程指南

C# 关键字

params（C# 参考）

使用 **params** 关键字可以指定采用数目可变的参数的方法参数。

可以发送参数声明中所指定类型的逗号分隔的参数列表或指定类型的参数数组。还可以不发送参数。如果未发送任何参数，则 **params** 列表的长度为零。

在方法声明中的 **params** 关键字之后不允许任何其他参数，并且在方法声明中只允许一个 **params** 关键字。

下面的示例演示可向 **params** 参数发送参数的各种方法。

```
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank li
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being cal
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);
```

```
        // The following call causes a compiler error because the o
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the enti
        // integer array becomes the first element of the params ar
        UseParams2(myIntArray);
    }
}
/*
Output:
    1 2 3 4
    1 a test

    5 6 7 8 9
    2 b test again
    System.Int32[]
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

方法参数（C# 参考）

ref（C# 参考）

ref 关键字通过引用（而非值）传递参数。通过引用传递的效果是，对所调用方法中的参数进行的任何更改都反映在调用方法中。例如，如果调用方传递本地变量表达式或数组元素访问表达式，所调用方法会将对象替换为 ref 参数引用的对象，然后调用方的本地变量或数组元素将开始引用新对象。

> 注意
>
> 不要混淆通过引用传递的概念与引用类型的概念。这两种概念是不同的。无论方法参数是值类型还是引用类型，均可由 **ref** 修改。当通过引用传递时，不会对值类型装箱。

若要使用 **ref** 参数，方法定义和调用方法均必须显式使用 **ref** 关键字，如下面的示例所示。

```
class RefExample
{
    static void Method(ref int i)
    {
        // Rest the mouse pointer over i to verify that it is an ir
        // The following statement would cause a compiler error if
        // were boxed as an object.
        i = i + 44;
    }

    static void Main()
    {
        int val = 1;
        Method(ref val);
        Console.WriteLine(val);

        // Output: 45
    }
}
```

传递到 **ref** 形参的实参必须先经过初始化，然后才能传递。这与 **out** 形参不同，在传递之前，不需要显式初始化该形参的实参。有关详细信息，请参阅 out。

类的成员不能具有仅在 **ref** 和 **out** 方面不同的签名。如果类型的两个成员之间的唯一区别在于其中一个具有 **ref** 参数，而另一个具有 **out** 参数，则会发生编译错误。例如，以下代码将不会编译。

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

但是，当一个方法具有 **ref** 或 **out** 参数，另一个方法具有值参数时，则可以完成重载，如下面的示例所示。

```
class RefOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}
```

在其他要求签名匹配的情况下（如隐藏或重写），**ref** 和 **out** 是签名的一部分，相互之间不匹配。

属性不是变量。它们是方法，不能传递到 **ref** 参数。

有关如何传递数组的信息，请参阅使用 ref 和 out 传递数组（C# 编程指南）。

你不能将 **ref** 和 **out** 关键字用于以下几种方法：

- 异步方法，通过使用 async 修饰符定义。
- 迭代器方法，包括 yield return 或 **yield break** 语句。

前面的示例演示当通过引用传递值类型时会发生什么情况。你还可以使用 **ref** 关键字传递引用类型。通过引用传递引用类型可以使所调用方法将调用方法中的对象替换为引用参数所引用的对象。对象的存储位置按引用参数的值传递到方法。如果更改参数存储位置中的值（以指向新对象），你还可以将存储位置更改为调用方所引用的位置。下面的示例将引用类型的实例作为 **ref** 参数传递。有关如何通过值和引用传递引用类型的详细信息，请参阅传递引用类型参数（C# 编程指南）。

```
class RefExample2
{
    static void ChangeByReference(ref Product itemRef)
    {
        // The following line changes the address that is stored ir
        // parameter itemRef. Because itemRef is a ref parameter, t
        // address that is stored in variable item in Main also is
        itemRef = new Product("Stapler", 99999);

        // You can change the value of one of the properties of
        // itemRef. The change happens to item in Main as well.
        itemRef.ItemID = 12345;
    }

    static void Main()
    {
        // Declare an instance of Product and display its initial v
        Product item = new Product("Fasteners", 54321);
        System.Console.WriteLine("Original values in Main.  Name: 
            item.ItemName, item.ItemID);

        // Send item to ChangeByReference as a ref argument.
        ChangeByReference(ref item);
        System.Console.WriteLine("Back in Main.  Name: {0}, ID: {1]
            item.ItemName, item.ItemID);
    }
}

class Product
{
    public Product(string name, int newID)
    {
        ItemName = name;
        ItemID = newID;
    }

    public string ItemName { get; set; }
    public int ItemID { get; set; }
}

// Output:
//Original values in Main.  Name: Fasteners, ID: 54321

//Back in Main.  Name: Stapler, ID: 12345
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

传递参数（C# 编程指南）

方法参数（C# 参考）

C# 关键字

out（C# 参考）

你可以在两个上下文（每个都是指向详细信息的链接）中使用 **out** 上下文关键字作为参数修饰符，或在接口和委托中使用泛型类型参数声明。本主题讨论参数修饰符，但你可以参阅其他主题了解关于泛型类型参数声明的信息。

out 关键字通过引用传递参数。这与 ref 关键字相似，只不过 **ref** 要求在传递之前初始化变量。若要使用 **out** 参数，方法定义和调用方法均必须显式使用 **out** 关键字。例如：

```
class OutExample
{
    static void Method(out int i)
    {
        i = 44;
    }
    static void Main()
    {
        int value;
        Method(out value);
        // value is now 44
    }
}
```

尽管作为 **out** 参数传递的变量无需在传递之前初始化，调用方法仍要求在方法返回之前赋值。

尽管 **ref** 和 **out** 关键字会导致不同的运行时行为，它们并不被视为编译时方法签名的一部分。因此，如果唯一的不同是一个方法采用 **ref** 参数，而另一个方法采用 **out** 参数，则无法重载这两个方法。例如，以下代码将不会编译：

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

但是，如果一个方法采用 **ref** 或 **out** 参数，而另一个方法采用其他参数，则可以完成重载，如：

```
class OutOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(out int i) { i = 5; }
}
```

属性不是变量，因此不能作为 **out** 参数传递。

有关传递数组的信息，请参阅使用 ref 和 out 传递数组（C# 编程指南）。

你不能将 **ref** 和 **out** 关键字用于以下几种方法：

- 异步方法，通过使用 async 修饰符定义。
- 迭代器方法，包括 yield return 或 **yield break** 语句。

如果希望方法返回多个值，可以声明 **out** 方法。下面的示例使用 **out** 返回具有单个方法调用的三个变量。注意，第三个参数赋 null 值。这使得方法可以有选择地返回值。

```
class OutReturnExample
{
    static void Method(out int i, out string s1, out string s2)
    {
        i = 44;
        s1 = "I've been returned";
        s2 = null;
    }
    static void Main()
    {
        int value;
        string str1, str2;
        Method(out value, out str1, out str2);
        // value is now 44
        // str1 is now "I've been returned"
        // str2 is (still) null;
    }
}
```

请参阅

C# 参考

out 参数修饰符（C# 参考）

out 关键字会导致参数通过引用来传递。这与 ref 关键字类似，不同之处在于 **ref** 要求变量必须在传递之前进行初始化。若要使用 **out** 参数，方法定义和调用方法都必须显式使用 **out** 关键字。例如：

```
class OutExample
{
    static void Method(out int i)
    {
        i = 44;
    }
    static void Main()
    {
        int value;
        Method(out value);
        // value is now 44
    }
}
```

尽管作为 **out** 参数传递的变量不必在传递之前进行初始化，但被调用的方法需要在返回之前赋一个值。

尽管 **ref** 和 **out** 关键字会导致不同的运行时行为，但在编译时并不会将它们视为方法签名的一部分。因此，如果两个方法唯一的区别是：一个接受 **ref** 参数，另一个接受 **out** 参数，则无法重载这两个方法。例如，不会编译下面的代码：

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

但是，如果一个方法采用 **ref** 或 **out** 参数，而另一个方法不采用这两类参数，则可以进行重载，如下所示：

```
class OutOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(out int i) { i = 5; }
}
```

属性不是变量，因此不能作为 **out** 参数传递。

有关传递数组的信息，请参见使用 ref 和 out 传递数组（C# 编程指南）。

不能为以下方法使用 **ref** 和 **out** 关键字:

- 异步方法，或者使用 async 修饰符，定义。
- 迭代器方法，包括一个 将返回 或 **yield break** 语句。

当希望方法返回多个值时，声明 **out** 方法很有用。下面的示例使用 **out** 在一次方法调用中返回三个变量。请注意，第三个参数所赋的值为 Null。这样使方法可以有选择地返回值。

```
class OutReturnExample
{
    static void Method(out int i, out string s1, out string s2)
    {
        i = 44;
        s1 = "I've been returned";
        s2 = null;
    }
    static void Main()
    {
        int value;
        string str1, str2;
        Method(out value, out str1, out str2);
        // value is now 44
        // str1 is now "I've been returned"
        // str2 is (still) null;
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

方法参数（C# 参考）

命名空间关键字（C# 参考）

本节描述与 using 命名空间关联的关键字和运算符：

- 命名空间
- using
- .运算符
- :: 运算符
- 外部别名

请参阅

C# 参考

C# 编程指南

C# 关键字

命名空间（C# 编程指南）

命名空间（C# 参考）

namespace 关键字用于声明一组相关对象的大小。可以使用命名空间组织代码元素和创建全局唯一类型。

```
namespace SampleNamespace
{
    class SampleClass { }

    interface SampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace SampleNamespace.Nested
    {
        class SampleClass2 { }
    }
}
```

备注

在一个命名空间中，可以声明一个或多个下列类型：

- 另一个命名空间
- class
- interface
- struct
- enum
- Delegate — 委托

无论您是否在 C# 源文件中显式声明了命名空间，编译器都会添加一个默认的命名空间。该未命名的命名空间（有时称为全局命名空间）存在于每一个文件中。全局命名空间中的任何标识符都可用于命名的命名空间中。

命名空间隐式具有公共访问权，并且这是不可修改的。有关可以分配给命名空间中的元素的访问修饰符的讨论，请参见访问修饰符（C# 参考）。

在两个或更多的声明中定义一个命名空间是可以的。例如，下面的示例将两个类定义为 MyCompany 命名空间的一部分：

```
namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}
```

下面的示例显示了如何在嵌套的命名空间中调用静态方法。

```
namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}
// Output: Hello
```

更多信息

有关使用命名空间的更多信息，请参见下列主题：

- 命名空间（C# 编程指南）

- 使用命名空间（C# 编程指南）
- 如何：使用全局命名空间别名（C# 编程指南）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

命名空间关键字（C# 参考）

using（C# 参考）

using（C# 参考）

using 关键字有两个主要用途：

- 作为指令，用于为命名空间创建别名或导入其他命名空间中定义的类型。请参见 using 指令。
- 作为语句，用于定义一个范围，在此范围的末尾将释放对象。请参见 using 语句。

请参阅

C# 参考

C# 编程指南

C# 关键字

命名空间关键字（C# 参考）

命名空间（C# 编程指南）

extern（C# 参考）

using 指令（C# 参考）

using 指令有三种用途：

- 允许在命名空间中使用类型，这样无需在该命名空间中限定某个类型的使用：

```
using System.Text;
```

- 允许访问类型的静态成员，而无需限定使用类型名称进行访问：

```
using static System.Math;
```

- 为命名空间或类型创建别名。这称为 *using* 别名指令。

```
using Project = PC.MyCompany.Project;
```

using 关键字还用于创建 *using* 语句，此类语句有助于确保正确处理 IDisposable 对象（如文件和字体）。有关详细信息，请参阅 using 语句。

Using Static 类型

你可以访问类型的静态成员，而无需限定使用类型名称进行访问：

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

Using static 仅导入可访问的静态成员和指定类型中声明的嵌套类型。不导入继承的成员。可以从任何带 using static 指令的已命名类型导入，包括 Visual Basic 模块。如果 F# 顶级函数在元数据中显示为一个已命名类型（其名称是有效的 C# 标识符）的静态成员，则可以导入该 F# 函数。

Using static 使指定类型中声明的扩展方法可用于扩展方法查找。但是，扩展方法的名称不导入到代码中非限定引用的作用域中。

同一编译单元或命名空间中通过不同 using static 命令从不同类型导入的具有相同名称方法组成一个方法组。这些方法组内的重载解决方法遵循一般 C# 规则。

备注

using 指令的范围限于显示它的文件。

创建 **using** 别名，以便更易于将标识符限定为命名空间或类型。using 别名指令的右侧必须始终是一个完全限定类型，而与前面的 using 指令无关。

创建 **using** 指令，以便在命名空间中使用类型而不必指定命名空间。 **using** 指令不为你提供对嵌套在指定命名空间中的任何命名空间的访问权限。

命名空间分为两类：用户定义的命名空间和系统定义的命名空间。用户定义的命名空间是在代码中定义的命名空间。有关系统定义的命名空间的列表，请参阅 .NET Framework 类库。

有关引用其他程序集中的方法的示例，请参阅创建和使用 C# DLL。

示例 1

说明

下面的示例显示如何为命名空间定义和使用 **using** 别名：

代码

```
namespace PC
{
    // Define an alias for the nested namespace.
    using Project = PC.MyCompany.Project;
    class A
    {
        void M()
        {
            // Use the alias
            Project.MyClass mc = new Project.MyClass();
        }
    }
    namespace MyCompany
    {
        namespace Project
        {
            public class MyClass { }
        }
    }
}
```

批注

using 别名指令的右侧不能有开放式泛型类型。例如，不能为 List<T> 创建 using 别名，但可以为 List<int> 创建。

示例 2

说明

下面的示例显示如何为类定义 **using** 指令和 **using** 别名：

代码

```
using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    public class MyClass
    {
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }

}

namespace NameSpace2
{
    class MyClass<T>
    {
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
}

namespace NameSpace3
{
    // Using directive:
    using NameSpace1;
```

```
    // Using directive:
    using NameSpace2;

    class MainClass
    {
        static void Main()
        {
            AliasToMyClass instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            UsingAlias instance2 = new UsingAlias();
            Console.WriteLine(instance2);

        }
    }
}
// Output:
//    You are in NameSpace1.MyClass.
//    You are in NameSpace2.MyClass.
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

使用命名空间（C# 编程指南）

C# 关键字

命名空间关键字（C# 参考）

命名空间（C# 编程指南）

using 语句（C# 参考）

using 语句（C# 参考）

提供能确保正确使用 IDisposable 对象的方便语法。

下面的示例演示如何使用 using 语句。

```
using (Font font1 = new Font("Arial", 10.0f))
{
    byte charset = font1.GdiCharSet;
}
```

备注

File 和 Font 是访问非托管资源（本例中为文件句柄和设备上下文）的托管类型的示例。有许多其他类别的非托管资源和封装这些资源的类库类型。所有这些类型都必须实现 IDisposable 接口。

按照规则，当使用 **IDisposable** 对象时，应在 **using** 语句中声明和实例化此对象。using 语句按照正确的方式调用对象上的 Dispose 方法，并（在您按照前面所示方式使用它时）会导致在调用 Dispose 时对象自身离开作用域。 在 using 块中，对象是只读的并且无法修改或重新赋值。

using 语句确保即使在调用对象上的方法时发生异常Dispose方法也会被调用。 可通过将对象放入 try 块中并在finally 块中调用Dispose来达到同样的结果；实际上，这就是编译器转换 using 语句的方式。 前面的代码示例在编译时将扩展为以下代码（请注意，使用额外的大括号为对象创建限制范围）：

```
{
  Font font1 = new Font("Arial", 10.0f);
  try
  {
    byte charset = font1.GdiCharSet;
  }
  finally
  {
    if (font1 != null)
      ((IDisposable)font1).Dispose();
  }
}
```

如下面的示例所示，可以在 **using** 语句中声明一个类型的多个实例。

```
using (Font font3 = new Font("Arial", 10.0f),
            font4 = new Font("Arial", 10.0f))
{
    // Use font3 and font4.
}
```

可以实例化资源对象，然后将变量传递给 **using** 语句，但这不是最佳做法。在这种情况下，该对象将在控制权离开 **using** 块之后保持在范围内，即使它可能将不再具有对其非托管资源的访问权也是如此。 换句话说，它将不再是完全初始化的。 如果尝试在 **using** 块外部使用该对象，则可能导致引发异常。 由于这个原因，通常最好是在 using 语句中实例化该对象并将其作用域限制到using块中。

```
Font font2 = new Font("Arial", 10.0f);
using (font2) // not recommended
{
    // use font2
}
// font2 is still in scope
// but the method call throws an exception
float f = font2.GetHeight();
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

using 指令（C# 参考）

Garbage Collection

Implementing a Dispose Method

外部别名（C# 参考）

可能必须引用两个具有相同完全限定类型名的程序集版本。例如，可能必须在同一应用程序中使用程序集的两个或多个版本。通过使用外部程序集别名，可以将来自每个程序集的命名空间包装在由别名命名的根级别命名空间中，从而使这些命名空间可以在同一文件中使用。

> 注意
>
> extern 关键字还用作方法修饰符，声明用非托管代码编写的方法。

若要引用两个具有相同完全限定类型名的程序集，必须在命令提示符下指定别名，如下所示：

/r:GridV1=grid.dll

/r:GridV2=grid20.dll

这将创建外部别名 GridV1 和 GridV2。若要从程序中使用这些别名，请使用 **extern** 关键字引用它们。例如：

extern alias GridV1;

extern alias GridV2;

每一个外部别名声明都引入一个额外的根级别命名空间，它与全局命名空间平行，而不是在全局命名空间内。因此，通过使用根源于相应命名空间别名的完全限定名，可以无歧义地引用每个程序集的类型。

在上面的示例中，GridV1::Grid 是来自 grid.dll 的网格控件，而 GridV2::Grid 是来自 grid20.dll 的网格控件。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

命名空间关键字（C# 参考）

:: 运算符（C# 参考）

/reference (C# Compiler Options)

运算符关键字（C# 参考）

用于执行其他操作 (如创建对象，检查对象的运行时类型，获取类型的大小和其他操作。本节介绍下列关键字：

- as 将对象转换为可兼容类型。
- 等待 挂起异步方法，直到其中一等待任务完成。
- is 检查对象的运行时类型。
- new
  - new 运算符 创建对象。
  - new 修饰符 隐藏继承成员。
  - new 约束 限定类型参数。
- sizeof 获取类型的大小。
- typeof 获取类型的 **System.Type** 对象。
- true
  - true 运算符 返回布尔值 true 表示真，否则返回 false。
  - true 表示布尔值 true。
- false
  - false 运算符 返回布尔值 true 表示假，否则返回 false。
  - false 表示布尔值 false。
- stackalloc 在堆栈上分配内存块。

在语句一节中介绍了下列可用作运算符和语句的关键字：

- checked 指定已检查的上下文。
- unchecked 指定未检查的上下文。

请参阅

C# 参考

C# 编程指南

C# 关键字

C# 运算符

as（C# 参考）

可以使用 **as** 运算符执行转换的某些类型在兼容之间的引用类型或 可以为 null 的类型。下面的代码提供了一个示例。

```
class csrefKeywordsOperators
{
    class Base
    {
        public override string  ToString()
        {
           return "Base";
        }
    }
    class Derived : Base
    { }

    class Program
    {
        static void Main()
        {

            Derived d = new Derived();

            Base b = d as Base;
            if (b != null)
            {
                Console.WriteLine(b.ToString());
            }

        }
    }
}
```

备注

as 运算符类似于强制转换操作。但是，因此，如果转换是不可能的，**as** 返回 **null** 而不引发异常。请看下面的示例：

```
expression as type
```

代码与下面的表达式是等效的，但 expression 变量只计算一次。

```
expression is type ? (type)expression : (type)null
```

请注意 **as** 运算符执行只引用转换、nullable 转换和装箱转换。 **as** 运算符不能执行其他转换，如用户定义的转换，应是通过使用转换的表达式。

```
class ClassA { }
class ClassB { }

class MainClass
{
    static void Main()
    {
        object[] objArray = new object[6];
        objArray[0] = new ClassA();
        objArray[1] = new ClassB();
        objArray[2] = "hello";
        objArray[3] = 123;
        objArray[4] = 123.4;
        objArray[5] = null;

        for (int i = 0; i < objArray.Length; ++i)
        {
            string s = objArray[i] as string;
            Console.Write("{0}:", i);
            if (s != null)
            {
                Console.WriteLine("'" + s + "'");
            }
            else
            {
                Console.WriteLine("not a string");
            }
        }
    }
}
/*
Output:
0:not a string
1:not a string
2:'hello'
3:not a string
4:not a string
5:not a string
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

is（C# 参考）

?: 运算符（C# 参考）

运算符关键字（C# 参考）

await（C# 参考）

await 运算符在异步方法应用于任务，以挂起执行方法，直到所等待的任务完成。任务表示正在进行的工作。

在其中使用 **await** 的异步方法必须通过 async 关键字进行修改。使用 **async** 修饰符定义并且通常包含一个或多个 **await** 表达式的这类方法称为异步方法。

注意

async 和 **await** 关键字是在 Visual Studio 2012 中引入的。有关异步编程的说明，请参阅使用 Async 和 Await 的异步编程（C# 和 Visual Basic）。

应用 **await** 运算符的任务通常是实现基于任务的异步模式的方法调用的返回值。示例包括 Task 或 Task<TResult> 类型的值。

在以下代码中，HttpClient 方法 GetByteArrayAsync 返回 **Task<byte[]>** (getContentsTask)。当任务完成时，任务约定生成实际字节数组。 **await** 运算符应用于 getContentsTask 以在 SumPageSizesAsync 中挂起执行，直到 getContentsTask 完成。同时，控制权会返回给 SumPageSizesAsync 的调用方。当 getContentsTask 完成之后，**await** 表达式计算为字节数组。

```
private async Task SumPageSizesAsync()
{
    // To use the HttpClient type in desktop apps, you must include
    // reference for the System.Net.Http namespace.
    HttpClient client = new HttpClient();
    // . . .
    Task<byte[]> getContentsTask = client.GetByteArrayAsync(url);
    byte[] urlContents = await getContentsTask;

    // Equivalently, now that you see how it works, you can write t
    //byte[] urlContents = await client.GetByteArrayAsync(url);
    // . . .
}
```

重要事项

有关完整示例，请参阅演练：使用 Async 和 Await 访问 Web（C# 和 Visual Basic）。可以从 Microsoft 网站上的开发人员代码示例下载该示例。该示例处于 AsyncWalkthrough_HttpClient 项目中。

如前面的示例中所示，如果 **await** 应用于返回 **Task<TResult>** 的方法调用结果，则 **await** 表达式的类型为 TResult。如果 **await** 应用于返回 **Task** 的方法调用结果，则 **await** 表达式的类型为 void。以下示例演示了差异。

```
// Keyword await used with a method that returns a Task<TResult>.
TResult result = await AsyncMethodThatReturnsTaskTResult();

// Keyword await used with a method that returns a Task.
await AsyncMethodThatReturnsTask();
```

await 表达式不阻止正在执行它的线程。而是使编译器将剩下的异步方法注册为等待任务的延续任务。控制权随后会返回给异步方法的调用方。任务完成时，它会调用其延续任务，异步方法的执行会在暂停的位置处恢复。

await 表达式只能在由 **async** 修饰符标记的立即封闭方法体、lambda 表达式或异步方法中出现。术语“*await*”在该上下文中仅用作关键字。在其他位置，它会解释为标识符。在方法、lambda 表达式或匿名方法中，**await** 表达式不能在同步函数体、查询表达式、lock 语句块或不安全上下文中出现。

异常

大多数异步方法返回 Task 或 Task<TResult>。返回任务的属性携带有关其状态和历史记录的信息，如任务是否完成、异步方法是否导致异常或已取消以及最终结果是什么。 **await** 运算符可访问这些属性。

如果等待的任务返回异步方法导致异常，则 **await** 运算符会重新引发异常。

如果等待的任务返回异步方法取消，则 **await** 运算符会重新引发 OperationCanceledException。

处于故障状态的单个任务可以反映多个异常。例如，任务可能是对 Task.WhenAll 调用的结果。等待此类任务时，等待操作仅重新引发异常之一。但是，无法预测重新引发的异常。

有关异步方法中的错误处理的示例，请参阅 try-catch（C# 参考）。

下面的 Windows 窗体示例阐释如何在异步方法 WaitAsynchronouslyAsync 中使用 **await**。将该方法的行为与 WaitSynchronously 的行为进行对比。如果未向任务应用 **await** 运算符，WaitSynchronously 就会同步运行，而不管其定义中是否使用了 **async** 修饰符和在主体中是否调用了 Thread.Sleep。

```
private async void button1_Click(object sender, EventArgs e)
{
    // Call the method that runs asynchronously.
    string result = await WaitAsynchronouslyAsync();

    // Call the method that runs synchronously.
    //string result = await WaitSynchronously ();

    // Display the result.
    textBox1.Text += result;
}

// The following method runs asynchronously. The UI thread is not
// blocked during the delay. You can move or resize the Form1 windo
// while Task.Delay is running.
public async Task<string> WaitAsynchronouslyAsync()
{
    await Task.Delay(10000);
    return "Finished";
}

// The following method runs synchronously, despite the use of asyn
// You cannot move or resize the Form1 window while Thread.Sleep
// is running because the UI thread is blocked.
public async Task<string> WaitSynchronously()
{
    // Add a using directive for System.Threading.
    Thread.Sleep(10000);
    return "Finished";
}
```

请参阅

使用 Async 和 Await 的异步编程（C# 和 Visual Basic）

演练：使用 Async 和 Await 访问 Web（C# 和 Visual Basic）

async（C# 参考）

is（C# 参考）

检查对象是否与给定类型兼容。例如，下面的代码可以确定对象是否为 **MyObject** 类型的一个实例，或者对象是否为从 **MyObject** 派生的一个类型：

```
if (obj is MyObject)
{
}
```

如果所提供的表达式非空，并且所提供的对象可以强制转换为所提供的类型而不会导致引发异常，则 **is** 表达式的计算结果将是 **true**。

如果已知表达式将始终是 **true** 或始终是 **false**，则 **is** 关键字将导致编译时警告，但是，通常在运行时才计算类型兼容性。

不能重载 **is** 运算符。

请注意，**is** 运算符只考虑引用转换、装箱转换和取消装箱转换。不考虑其他转换，如用户定义的转换。

在 **is** 运算符的左侧不允许使用匿名方法。lambda 表达式属于例外。

```
class Class1 {}
class Class2 {}
class Class3 : Class2 { }

class IsTest
{
    static void Test(object o)
    {
        Class1 a;
        Class2 b;

        if (o is Class1)
        {
            Console.WriteLine("o is Class1");
            a = (Class1)o;
            // Do something with "a."
        }
        else if (o is Class2)
        {
            Console.WriteLine("o is Class2");
            b = (Class2)o;
            // Do something with "b."
        }

        else
        {
            Console.WriteLine("o is neither Class1 nor Class2.");
        }
    }
    static void Main()
    {
        Class1 c1 = new Class1();
        Class2 c2 = new Class2();
        Class3 c3 = new Class3();
        Test(c1);
        Test(c2);
        Test(c3);
        Test("a string");
    }
}
/*
Output:
o is Class1
o is Class2
o is Class2
o is neither Class1 nor Class2.
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

typeof（C# 参考）

as（C# 参考）

运算符关键字（C# 参考）

new（C# 参考）

在 C# 中，**new** 关键字可用作运算符、修饰符或约束。

new 运算符

用于创建对象和调用构造函数。

new 修饰符

用于隐藏基类中被继承的成员。

new 约束

用于在泛型声明中约束可能用作类型参数的参数的类型。

请参阅

C# 参考

C# 编程指南

C# 关键字

new 运算符（C# 参考）

用于创建对象和调用构造函数。例如：

```
Class1 obj  = new Class1();
```

还可用于创建匿名类型的实例：

```
var query = from cust in customers
            select new {Name = cust.Name, Address = cust.PrimaryAdd
```

new 运算符还用于调用值类型的默认构造函数。例如：

```
int i = new int();
```

在上一个语句中，i 初始化为 0，它是 **int** 类型的默认值。该语句的效果等同于：

```
int i = 0;
```

有关默认值的完整列表，请参见默认值表（C# 参考）。

请记住，为结构声明默认的构造函数是错误的，因为每一个值类型都隐式具有一个公共的默认构造函数。可以在结构类型上声明参数化构造函数以设置其初始值，但是，只有在需要默认值之外的值时才必须这样做。

值类型对象（例如结构）是在堆栈上创建的，而引用类型对象（例如类）是在堆上创建的。两种类型的对象都是自动销毁的，但是，基于值类型的对象是在超出范围时销毁，而基于引用类型的对象则是在对该对象的最后一个引用被移除之后在某个不确定的时间销毁。对于占用固定资源（例如大量内存、文件句柄或网络连接）的引用类型，有时需要使用确定性终止以确保对象被尽快销毁。有关更多信息，请参见 using 语句（C# 参考）。

不能重载 **new** 运算符。

如果 **new** 运算符分配内存失败，将引发异常 OutOfMemoryException。

在下面的示例中，通过使用 **new** 运算符创建并初始化一个 **struct** 对象和一个类对象，然后为它们赋值。显示了默认值和所赋的值。

```
struct SampleStruct
{
   public int x;
```

```
    public int y;

    public SampleStruct(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

class SampleClass
{
    public string name;
    public int id;

    public SampleClass() {}

    public SampleClass(int id, string name)
    {
        this.id = id;
        this.name = name;
    }
}

class ProgramClass
{
    static void Main()
    {
        // Create objects using default constructors:
        SampleStruct Location1 = new SampleStruct();
        SampleClass Employee1 = new SampleClass();

        // Display values:
        Console.WriteLine("Default values:");
        Console.WriteLine("   Struct members: {0}, {1}",
               Location1.x, Location1.y);
        Console.WriteLine("   Class members: {0}, {1}",
               Employee1.name, Employee1.id);

        // Create objects using parameterized constructors:
        SampleStruct Location2 = new SampleStruct(10, 20);
        SampleClass Employee2 = new SampleClass(1234, "Cristina Potra

        // Display values:
        Console.WriteLine("Assigned values:");
        Console.WriteLine("   Struct members: {0}, {1}",
               Location2.x, Location2.y);
        Console.WriteLine("   Class members: {0}, {1}",
               Employee2.name, Employee2.id);
    }
}
/*
Output:
Default values:
```

```
    Struct members: 0, 0
    Class members: , 0
 Assigned values:
    Struct members: 10, 20
    Class members: Cristina Potra, 1234
 */
```

注意，在示例中字符串的默认值为 **null**，因此未显示它。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

运算符关键字（C# 参考）

new（C# 参考）

匿名类型（C# 编程指南）

new 修饰符（C# 参考）

在用作声明修饰符时，**new** 关键字可以显式隐藏从基类继承的成员。隐藏继承的成员时，该成员的派生版本将替换基类版本。虽然可以不使用 **new** 修饰符来隐藏成员，但将收到编译器警告。如果使用 **new** 来显式隐藏成员，将禁止此警告。

若要隐藏继承的成员，请使用相同名称在派生类中声明该成员，并使用 **new** 修饰符对其进行修饰。例如:

```
public class BaseC
{
    public int x;
    public void Invoke() { }
}
public class DerivedC : BaseC
{
    new public void Invoke() { }
}
```

在此示例中，使用 BaseC.Invoke 隐藏了 DerivedC.Invoke。字段 x 不受影响，因为未使用类似名称将其隐藏。

通过继承隐藏名称采用下列形式之一：

- 通常，在类或结构中引入的常数、字段、属性或类型会隐藏与其共享名称的所有基类成员。有三种特殊情况。例如，如果将名称为 N 的新字段声明为不可调用的类型，并且基类型将 N 声明为一种方法，则新字段在调用语法中不会隐藏基声明。请参阅 C# 语言规范获取详细信息（请参阅“表达式”一节中“成员查找”部分）。
- 类或结构中引入的方法会隐藏基类中共享该名称的属性、字段和类型。它还会隐藏具有相同签名的所有基类方法。
- 类或结构中引入的索引器会隐藏具有相同签名的所有基类索引器。

对同一成员同时使用 **new** 和 override 是错误的做法，因为这两个修饰符的含义互斥。 **new** 修饰符会用同样的名称创建一个新成员并使原始成员变为隐藏。**override** 修饰符会扩展继承成员的实现。

在不隐藏继承成员的声明中使用 **new** 修饰符将会生成警告。

在此示例中，基类 BaseC 和派生类 DerivedC 使用相同的字段名 x，从而隐藏了继承字段的值。此示例演示 **new** 修饰符的用法。另外还演示了如何使用完全限定名访问基类的隐藏成员。

```
public class BaseC
{
    public static int x = 55;
    public static int y = 22;
}

public class DerivedC : BaseC
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseC.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
/*
Output:
100
55
22
*/
```

在此示例中，嵌套类隐藏了基类中同名的类。此示例演示如何使用 **new** 修饰符来消除警告消息，以及如何使用完全限定名来访问隐藏的类成员。

```
public class BaseC
{
    public class NestedC
    {
        public int x = 200;
        public int y;
    }
}

public class DerivedC : BaseC
{
    // Nested type hiding the base type members.
    new public class NestedC
    {
        public int x = 100;
        public int y;
        public int z;
    }

    static void Main()
    {
        // Creating an object from the overlapping class:
        NestedC c1  = new NestedC();

        // Creating an object from the hidden class:
        BaseC.NestedC c2 = new BaseC.NestedC();

        Console.WriteLine(c1.x);
        Console.WriteLine(c2.x);
    }
}
/*
Output:
100
200
*/
```

如果移除 **new** 修饰符，程序仍将编译和运行，但你会收到以下警告：

```
The keyword new is required on 'MyDerivedC.x' because it hides inhe
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

运算符关键字（C# 参考）

修饰符（C# 参考）

使用 Override 和 New 关键字进行版本控制（C# 编程指南）

了解何时使用 Override 和 New 关键字（C# 编程指南）

new 约束（C# 参考）

new 约束指定泛型类声明中的任何类型参数都必须有公共的无参数构造函数。如果要使用 new 约束，则该类型不能为抽象类型。

当泛型类创建类型的新实例，请将 **new** 约束应用于类型参数，如下面的示例所示：

```
class ItemFactory<T> where T : new()
{
    public T GetNewItem()
    {
        return new T();
    }
}
```

当与其他约束一起使用时，**new()** 约束必须最后指定：

```
public class ItemFactory2<T>
    where T : IComparable, new()
{
}
```

有关更多信息，请参见 类型参数的约束（C# 编程指南）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

System.Collections.Generic

C# 参考

C# 编程指南

C# 关键字

运算符关键字（C# 参考）

泛型（C# 编程指南）

sizeof（C# 参考）

用于获取非托管类型的大小（以字节为单位）。非托管类型包括下表列出的内置类型以及以下类型：

- 枚举类型
- 指针类型
- 用户定义的结构，不包含任何属于引用类型的字段或属性

下面的示例演示如何检索 **int** 的大小：

```
// Constant value 4:
int intSize = sizeof(int);
```

备注

从 C# 2.0 版开始，将 **sizeof** 应用于内置类型不再要求使用 unsafe 模式。

不能重载 **sizeof** 运算符。 **sizeof** 运算符的返回值是 **int** 类型。下表列出了一些常量值，这些值对应于以某些内置类型为操作数的 **sizeof** 表达式。

表达式	常量值
sizeof(sbyte)	1
sizeof(byte)	1
sizeof(short)	2
sizeof(ushort)	2
sizeof(int)	4
sizeof(uint)	4
sizeof(long)	8
sizeof(ulong)	8
sizeof(char)	2 (Unicode)
sizeof(float)	4
sizeof(double)	8
sizeof(decimal)	16
sizeof(bool)	1

对于所有其他类型（包括结构），**sizeof** 运算符只能在不安全代码块中使用。尽管可以使用 Marshal.SizeOf 方法，但此方法返回的值并不总是与 **sizeof** 返回的值相同。 Marshal.SizeOf 在封送类型后返回大小，而 **sizeof** 返回公共语言运行时分配的大小（包括所有填充）。

```
class MainClass
{
    // unsafe not required for primitive types
    static void Main()
    {
        Console.WriteLine("The size of short is {0}.", sizeof(short
        Console.WriteLine("The size of int is {0}.", sizeof(int));
        Console.WriteLine("The size of long is {0}.", sizeof(long)
    }
}
/*
Output:
    The size of short is 2.
    The size of int is 4.
    The size of long is 8.
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

运算符关键字（C# 参考）

enum（C# 参考）

不安全代码和指针（C# 编程指南）

结构（C# 编程指南）

常量（C# 编程指南）

typeof（C# 参考）

用于获取类型的 **System.Type** 对象。 **typeof** 表达式采用以下形式：

```
System.Type type = typeof(int);
```

备注

若要获取表达式的运行时类型，可以使用 .NET Framework 方法 GetType，如以下示例中所示：

```
int i = 0;
System.Type type = i.GetType();
```

不能重载 **typeof** 运算符。

typeof 运算符也能用于公开的泛型类型。具有不止一个类型参数的类型的规范中必须有适当数量的逗号。下面的示例演示如何确定方法的返回类型是否是泛型 IEnumerable<T>。假定此方法是 MethodInfo 类型的实例：

```
string s = method.ReturnType.GetInterface
    (typeof(System.Collections.Generic.IEnumerable<>).FullName);
```

```
public class ExampleClass
{
   public int sampleMember;
   public void SampleMethod() {}

   static void Main()
   {
      Type t = typeof(ExampleClass);
      // Alternatively, you could use
      // ExampleClass obj = new ExampleClass();
      // Type t = obj.GetType();

      Console.WriteLine("Methods:");
      System.Reflection.MethodInfo[] methodInfo = t.GetMethods();

      foreach (System.Reflection.MethodInfo mInfo in methodInfo)
         Console.WriteLine(mInfo.ToString());

      Console.WriteLine("Members:");
      System.Reflection.MemberInfo[] memberInfo = t.GetMembers();

      foreach (System.Reflection.MemberInfo mInfo in memberInfo)
         Console.WriteLine(mInfo.ToString());
   }
}
/*
 Output:
    Methods:
    Void SampleMethod()
    System.String ToString()
    Boolean Equals(System.Object)
    Int32 GetHashCode()
    System.Type GetType()
    Members:
    Void SampleMethod()
    System.String ToString()
    Boolean Equals(System.Object)
    Int32 GetHashCode()
    System.Type GetType()
    Void .ctor()
    Int32 sampleMember
*/
```

此示例使用 GetType 方法确定用来包含数值计算的结果的类型。这取决于结果数字的存储要求。

```
class GetTypeTest
{
    static void Main()
    {
        int radius = 3;
        Console.WriteLine("Area = {0}", radius * radius * Math.PI);
        Console.WriteLine("The type is {0}",
                          (radius * radius * Math.PI).GetType()
        );
    }
}
/*
Output:
Area = 28.2743338823081
The type is System.Double
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

System.Type

C# 参考

C# 编程指南

C# 关键字

is（C# 参考）

运算符关键字（C# 参考）

true（C# 参考）

用作重载运算符或文本：

true 运算符

true 标识符

请参阅

C# 参考

C# 编程指南

C# 关键字

true 运算符（C# 参考）

返回布尔值 **true** 以指示操作数为 true，否则返回 **false**。

在 C# 2.0 以前，**true** 和 **false** 运算符一直用来创建用户定义的可以为 null 的值类型，这些值类型与 **SqlBool** 等类型兼容。然而，现在该语言提供对可为 null 的值类型的内置支持，并且应该尽可能地使用它们而不是重载 **true** 和 **false** 运算符。有关更多信息，请参见 可以为 null 的类型（C# 编程指南）。

对于可以为 null 的布尔值，表达式 a != b 不一定等同于 !(a == b)，因为两个值之一或者全部都有可能为 null。需要单独重载 **true** 和 **false** 运算符，以便正确标识表达式中的 null 值。下面的示例演示如何重载及使用 **true** 和 **false** 运算符。

```
// For example purposes only. Use the built-in nullable bool
// type (bool?) whenever possible.
public struct DBBool
{
    // The three possible DBBool values.
    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);
    // Private field that stores –1, 0, 1 for False, Null, True.
    sbyte value;
    // Private instance constructor. The value parameter must be –1
    DBBool(int value)
    {
        this.value = (sbyte)value;
    }
    // Properties to examine the value of a DBBool. Return true if
    // DBBool has the given value, false otherwise.
    public bool IsNull { get { return value == 0; } }
    public bool IsFalse { get { return value < 0; } }
    public bool IsTrue { get { return value > 0; } }
    // Implicit conversion from bool to DBBool. Maps true to DBBool
    // false to DBBool.False.
    public static implicit operator DBBool(bool x)
    {
        return x ? True : False;
    }
    // Explicit conversion from DBBool to bool. Throws an exception
    // given DBBool is Null; otherwise returns true or false.
    public static explicit operator bool(DBBool x)
    {
        if (x.value == 0) throw new InvalidOperationException();
        return x.value > 0;
    }
    // Equality operator. Returns Null if either operand is Null; o
    // returns True or False.
    public static DBBool operator ==(DBBool x, DBBool y)
```

```
    {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value == y.value ? True : False;
    }
    // Inequality operator. Returns Null if either operand is Null;
    // returns True or False.
    public static DBBool operator !=(DBBool x, DBBool y)
    {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value != y.value ? True : False;
    }
    // Logical negation operator. Returns True if the operand is Fa
    // if the operand is Null, or False if the operand is True.
    public static DBBool operator !(DBBool x)
    {
        return new DBBool(-x.value);
    }
    // Logical AND operator. Returns False if either operand is Fal
    // Null if either operand is Null, otherwise True.
    public static DBBool operator &(DBBool x, DBBool y)
    {
        return new DBBool(x.value < y.value ? x.value : y.value);
    }
    // Logical OR operator. Returns True if either operand is True,
    // Null if either operand is Null, otherwise False.
    public static DBBool operator |(DBBool x, DBBool y)
    {
        return new DBBool(x.value > y.value ? x.value : y.value);
    }
    // Definitely true operator. Returns true if the operand is Tru
    // otherwise.
    public static bool operator true(DBBool x)
    {
        return x.value > 0;
    }
    // Definitely false operator. Returns true if the operand is Fa
    // otherwise.
    public static bool operator false(DBBool x)
    {
        return x.value < 0;
    }
    public override bool Equals(object obj)
    {
        if (!(obj is DBBool)) return false;
        return value == ((DBBool)obj).value;
    }
    public override int GetHashCode()
    {
        return value;
    }
    public override string ToString()
    {
        if (value > 0) return "DBBool.True";
```

```
            if (value < 0) return "DBBool.False";
            return "DBBool.Null";
        }
    }
```

重载 **true** 和 **false** 运算符的类型可以用于 if、do、while 和 for 语句以及条件表达式中的控制表达式。

如果类型定义了 **true** 运算符，它还必须定义 false 运算符。

类型不能直接重载条件逻辑运算符（&& 和 ||），但通过重载规则逻辑运算符和 **true** 与 **false** 运算符可以达到同样的效果。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

C# 运算符

false（C# 参考）

true 字面常数（C# 参考）

表示布尔值 true。

```
class TrueTest
{
    static void Main()
    {
        bool a = true;
        Console.WriteLine( a ? "yes" : "no" );
    }
}
/*
Output:
yes
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

false（C# 参考）

false（C# 参考）

用作重载运算符或文本：

- false 运算符
- false 标识符

请参阅

C# 参考

C# 编程指南

C# 关键字

false 运算符（C# 参考）

返回布尔值 **true** 以指示操作数为 **false**，否则返回 **false**。

在 C# 2.0 以前，**true** 和 **false** 运算符一直用来创建用户定义的可以为 null 的值类型，这些值类型与 **SqlBool** 等类型兼容。然而，现在该语言提供对可为 null 的值类型的内置支持，并且应该尽可能地使用它们而不是重载 **true** 和 **false** 运算符。有关更多信息，请参见 可以为 null 的类型（C# 编程指南）。

对于可以为 null 的布尔值，表达式 a != b 不一定等同于 !(a == b)，因为两个值之一或者全部都有可能为 null。必须单独重载 **true** 和 **false** 运算符，以便正确处理表达式中的 null 值。下面的示例演示如何重载及使用 **true** 和 **false** 运算符。

```
// For example purposes only. Use the built-in nullable bool
// type (bool?) whenever possible.
public struct DBBool
{
    // The three possible DBBool values.
    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);
    // Private field that stores –1, 0, 1 for False, Null, True.
    sbyte value;
    // Private instance constructor. The value parameter must be –1
    DBBool(int value)
    {
        this.value = (sbyte)value;
    }
    // Properties to examine the value of a DBBool. Return true if
    // DBBool has the given value, false otherwise.
    public bool IsNull { get { return value == 0; } }
    public bool IsFalse { get { return value < 0; } }
    public bool IsTrue { get { return value > 0; } }
    // Implicit conversion from bool to DBBool. Maps true to DBBool
    // false to DBBool.False.
    public static implicit operator DBBool(bool x)
    {
        return x ? True : False;
    }
    // Explicit conversion from DBBool to bool. Throws an exception
    // given DBBool is Null; otherwise returns true or false.
    public static explicit operator bool(DBBool x)
    {
        if (x.value == 0) throw new InvalidOperationException();
        return x.value > 0;
    }
    // Equality operator. Returns Null if either operand is Null; o
    // returns True or False.
    public static DBBool operator ==(DBBool x, DBBool y)
```

```
    {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value == y.value ? True : False;
    }
    // Inequality operator. Returns Null if either operand is Null;
    // returns True or False.
    public static DBBool operator !=(DBBool x, DBBool y)
    {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value != y.value ? True : False;
    }
    // Logical negation operator. Returns True if the operand is Fa
    // if the operand is Null, or False if the operand is True.
    public static DBBool operator !(DBBool x)
    {
        return new DBBool(-x.value);
    }
    // Logical AND operator. Returns False if either operand is Fal
    // Null if either operand is Null, otherwise True.
    public static DBBool operator &(DBBool x, DBBool y)
    {
        return new DBBool(x.value < y.value ? x.value : y.value);
    }
    // Logical OR operator. Returns True if either operand is True,
    // Null if either operand is Null, otherwise False.
    public static DBBool operator |(DBBool x, DBBool y)
    {
        return new DBBool(x.value > y.value ? x.value : y.value);
    }
    // Definitely true operator. Returns true if the operand is Tru
    // otherwise.
    public static bool operator true(DBBool x)
    {
        return x.value > 0;
    }
    // Definitely false operator. Returns true if the operand is Fa
    // otherwise.
    public static bool operator false(DBBool x)
    {
        return x.value < 0;
    }
    public override bool Equals(object obj)
    {
        if (!(obj is DBBool)) return false;
        return value == ((DBBool)obj).value;
    }
    public override int GetHashCode()
    {
        return value;
    }
    public override string ToString()
    {
        if (value > 0) return "DBBool.True";
```

```
            if (value < 0) return "DBBool.False";
            return "DBBool.Null";
        }
    }
```

重载 **true** 和 **false** 运算符的类型可以用于 if、do、while 和 for 语句以及条件表达式中的控制表达式。

如果类型定义了 **false** 运算符，则它还必须定义 true 运算符。

类型不能直接重载条件逻辑运算符 && 和 ||，但通过重载正则逻辑运算符以及运算符 **true** 与 **false** 可以达到同样的效果。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

C# 运算符

true（C# 参考）

false 字面常数（C# 参考）

表示布尔值 false。

```
class TestClass
{
    static void Main()
    {
        bool a = false;
        Console.WriteLine( a ? "yes" : "no" );
    }
}
// Output: no
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

true（C# 参考）

stackalloc（C# 参考）

stackalloc 关键字用于不安全的代码上下文中，以便在堆栈上分配内存块。

```
int* block = stackalloc int[100];
```

备注

关键字仅在局部变量的初始值中有效。下面的代码导致编译器错误。

```
int* block;
// The following assignment statement causes compiler errors. You
// can use stackalloc only when declaring and initializing a local
// variable.
block = stackalloc int[100];
```

由于涉及指针类型，因此 **stackalloc** 要求不安全上下文。有关更多信息，请参见不安全代码和指针（C# 编程指南）。

stackalloc 类似于 C 运行库中的 _alloca。

以下代码示例计算并演示 Fibonacci 序列中的前 20 个数字。每个数字是先前两个数字的和。在代码中，大小足够容纳 20 个 **int** 类型元素的内存块是在堆栈上分配的，而不是在堆上分配的。该块的地址存储在 fib 指针中。此内存不受垃圾回收的制约，因此不必将其钉住（通过使用 fixed）。内存块的生存期受限于定义它的方法的生存期。不能在方法返回之前释放内存。

```
class Test
{
    static unsafe void Main()
    {
        const int arraySize = 20;
        int* fib = stackalloc int[arraySize];
        int* p = fib;
        // The sequence begins with 1, 1.
        *p++ = *p++ = 1;
        for (int i = 2; i < arraySize; ++i, ++p)
        {
            // Sum the previous two numbers.
            *p = p[-1] + p[-2];
        }
        for (int i = 0; i < arraySize; ++i)
        {
            Console.WriteLine(fib[i]);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/*
Output
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
*/
```

安全性

不安全代码的安全性低于安全替代代码。但是，通过使用 **stackalloc** 可以自动启用公共语言运行时 (CLR) 中的缓冲区溢出检测功能。如果检测到缓冲区溢出，进程将尽快终止，以最大限度地减小执行恶意代码的机会。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

运算符关键字（C# 参考）

不安全代码和指针（C# 编程指南）

nameof（C# 和 Visual Basic 引用）

用于获取变量、类型或成员的简单（非限定）字符串名称。当报告代码中的错误、挂接“模型-视图-控制器”(MVC) 链接、触发属性更改事件等时，通常希望捕获方法的字符串名称。使用 **nameof** 有助于在重命名定义时使代码始终有效。在必须使用字符串来引用定义之前（在重命名代码元素时，此操作很脆弱，因为工具不知道要检查这些字符串）。

nameof 表达式具有此形式：

```
if (x == null) throw new ArgumentNullException(nameof(x));
WriteLine(nameof(person.Address.ZipCode)); // prints "ZipCode”
```

关键用例

这些示例显示 **nameof** 的关键用例。

验证参数：

```
void f(string s) {
    if (s == null) throw new ArgumentNullException(nameof(s));
}
```

MVC 操作链接：

<a>HTML

```
&lt;%= Html.ActionLink("Sign up",
             @typeof(UserController),
             @nameof(UserController.SignUp))
%&gt;
```

INotifyPropertyChanged：

```
int p {
    get { return this.p; }
    set { this.p = value; PropertyChanged(this, new PropertyChanged
}
```

XAML 依赖项属性：

```
public static DependencyProperty AgeProperty = DependencyProperty.F
```

日志记录：

```
void f(int i) {
    Log(nameof(f), "method entry");
}
```

特性:

```
[DebuggerDisplay("={" + nameof(GetString) + "()}")]
class C {
    string GetString() { }
}
```

示例

一些 C# 示例：

```
using Stuff = Some.Cool.Functionality
class C {
    static int Method1 (string x, int y) {}
    static int Method1 (string x, string y) {}
    int Method2 (int z) {}
    string f<T>() => nameof(T);
}

var c = new C()

nameof(C) -> "C"
nameof(C.Method1) -> "Method1"
nameof(C.Method2) -> "Method2"
nameof(c.Method1) -> "Method1"
nameof(c.Method2) -> "Method2"
nameof(z) -> "z" // inside of Method2 ok, inside Method1 is a compi
nameof(Stuff) = "Stuff"
nameof(T) -> "T" // works inside of method but not in attributes or
nameof(f) -> “f”
nameof(f<T>) -> syntax error
nameof(f<>) -> syntax error
nameof(Method2()) -> error “This expression does not have a name”
```

上面的很多示例都适用于 Visual Basic。下面是一些特定的 Visual Basic 示例：

VB

```
NameOf(a!Foo) -> ' error  "This expression does not have a name"
NameOf(dict("Foo")) -> ' error  "This expression does not have a na
NameOf(dict.Item("Foo")) -> ' error  "This expression does not have
NameOf(arr(2)) -> ' error  "This expression does not have a name":
Dim x = Nothing
NameOf(x.ToString(2)) -> ' error  "This expression does not have a
Dim o = Nothing
NameOf(o.Equals) -> ' result "Equals".  Warning: "Access of static
```

备注

nameof 的参数必须是简单名称、限定名称、成员访问、指定成员的基访问或指定成员的此类访问。参数表达式标识代码定义，但从不进行计算。

因为在语法上参数必须为表达式，因此有很多禁用内容无需列出。以下内容会产生错误，值得一提：预定义的类型（如 **int** 或 **void**）、可以为 null 的类型（Point?）、数组类型（Customer[,]）、指针类型 (Buffer*)、限定别名 (A::B)、未绑定的泛型类型 (Dictionary<,>)、预处理符号 (DEBUG) 和标签 (loop:)。

如果需要获取完全限定名，可以将 **typeof** 表达式和 **nameof**结合使用。

在这些示例中，显示了可使用类型名称并访问实例方法名称。按照计算表达式的要求，无需具有类型的实例。在某些情况下使用类型名称非常方便，因为只引用名称而不使用实例数据，因此不必构建实例变量或表达式。

你可以引用类中特性表达式的类成员。

没有任何方法可以获取租入“Method1 (str, str)”等签名信息。实现该操作的一种方法是使用表达式 Expression e = () => A.B.Method1("s1", "s2")，并从生成的表达式树中拉取 MemberInfo。

语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

有关详细信息，请参阅 Visual Basic 语言参考。

请参阅

C# 参考

C# 编程指南

typeof（C# 参考）

Visual Basic 语言参考

Visual Basic 编程指南

转换关键字（C# 参考）

本节描述在类型转换中使用的关键字：

- explicit
- implicit
- operator

请参阅

C# 参考

C# 编程指南

C# 关键字

explicit（C# 参考）

explicit 关键字用于声明必须使用强制转换来调用的用户定义的类型转换运算符。例如，在下面的示例中，此运算符将名为 Fahrenheit 的类转换为名为 Celsius 的类：

```
// Must be defined inside a class called Fahrenheit:
public static explicit operator Celsius(Fahrenheit fahr)
{
    return new Celsius((5.0f / 9.0f) * (fahr.degrees - 32));
}
```

可以如下所示调用此转换运算符：

```
Fahrenheit fahr = new Fahrenheit(100.0f);
Console.Write("{0} Fahrenheit", fahr.Degrees);
Celsius c = (Celsius)fahr;
```

转换运算符将源类型转换为目标类型。源类型提供转换运算符。与隐式转换不同，必须通过强制转换的方式来调用显式转换运算符。如果转换操作可能导致异常或丢失信息，则应将其标记为 **explicit**。这可以防止编译器无提示地调用可能产生无法预见后果的转换操作。

省略此强制转换将导致编译时错误 CS0266。

有关更多信息，请参见使用转换运算符（C# 编程指南）。

下面的示例提供 Fahrenheit 和 Celsius 类，它们中的每一个都为另一个提供显式转换运算符。

```
class Celsius
{
    public Celsius(float temp)
    {
        degrees = temp;
    }
    public static explicit operator Fahrenheit(Celsius c)
    {
        return new Fahrenheit((9.0f / 5.0f) * c.degrees + 32);
    }
    public float Degrees
    {
        get { return degrees; }
    }
    private float degrees;
}

class Fahrenheit
{
    public Fahrenheit(float temp)
    {
        degrees = temp;
    }
    // Must be defined inside a class called Fahrenheit:
    public static explicit operator Celsius(Fahrenheit fahr)
    {
        return new Celsius((5.0f / 9.0f) * (fahr.degrees - 32));
    }
    public float Degrees
    {
        get { return degrees; }
    }
    private float degrees;
}

class MainClass
{
    static void Main()
    {
        Fahrenheit fahr = new Fahrenheit(100.0f);
        Console.Write("{0} Fahrenheit", fahr.Degrees);
        Celsius c = (Celsius)fahr;

        Console.Write(" = {0} Celsius", c.Degrees);
        Fahrenheit fahr2 = (Fahrenheit)c;
        Console.WriteLine(" = {0} Fahrenheit", fahr2.Degrees);
    }
}
// Output:
// 100 Fahrenheit = 37.77778 Celsius = 100 Fahrenheit
```

下面的示例定义一个结构 Digit，该结构表示单个十进制数字。定义了一个运算符，用于将 byte 转换为 Digit，但因为并非所有字节都可以转换为 Digit，所以该转换是显式的。

```
struct Digit
{
    byte value;
    public Digit(byte value)
    {
        if (value > 9)
        {
            throw new ArgumentException();
        }
        this.value = value;
    }

    // Define explicit byte-to-Digit conversion operator:
    public static explicit operator Digit(byte b)
    {
        Digit d = new Digit(b);
        Console.WriteLine("conversion occurred");
        return d;
    }
}

class ExplicitTest
{
    static void Main()
    {
        try
        {
            byte b = 3;
            Digit d = (Digit)b; // explicit conversion
        }
        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}
/*
Output:
conversion occurred
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

implicit（C# 参考）

运算符（C# 参考）

如何：在结构间实现用户定义的转换（C# 编程指南）

链接的用户定义的显式转换在 C#

implicit（C# 参考）

implicit 关键字用于声明隐式的用户定义类型转换运算符。如果可以确保转换过程不会造成数据丢失，则可使用该关键字在用户定义类型和其他类型之间进行隐式转换。

```
class Digit
{
    public Digit(double d) { val = d; }
    public double val;
    // ...other members

    // User-defined conversion from Digit to double
    public static implicit operator double(Digit d)
    {
        return d.val;
    }
    //  User-defined conversion from double to Digit
    public static implicit operator Digit(double d)
    {
        return new Digit(d);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Digit dig = new Digit(7);
        //This call invokes the implicit "double" operator
        double num = dig;
        //This call invokes the implicit "Digit" operator
        Digit dig2 = 12;
        Console.WriteLine("num = {0} dig2 = {1}", num, dig2.val);
        Console.ReadLine();
    }
}
```

隐式转换可以通过消除不必要的类型转换来提高源代码的可读性。但是，因为隐式转换不需要程序员将一种类型显式强制转换为另一种类型，所以使用隐式转换时必须格外小心，以免出现意外结果。一般情况下，隐式转换运算符应当从不引发异常并且从不丢失信息，以便可以在程序员不知晓的情况下安全使用它们。如果转换运算符不能满足那些条件，则应将其标记为 **explicit**。有关更多信息，请参见使用转换运算符。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

explicit（C# 参考）

运算符（C# 参考）

如何：在结构间实现用户定义的转换（C# 编程指南）

运算符（C# 参考）

使用 **operator** 关键字来重载内置运算符，或提供类或结构声明中的用户定义转换。

下面是一个非常简化的表示分数的类。该类重载了 + 和 * 运算符，以执行分数加法和乘法；同时提供了将 Fraction 类型转换为 double 类型的转换运算符。

```
class Fraction
{
    int num, den;
    public Fraction(int num, int den)
    {
        this.num = num;
        this.den = den;
    }

    // overload operator +
    public static Fraction operator +(Fraction a, Fraction b)
    {
        return new Fraction(a.num * b.den + b.num * a.den,
           a.den * b.den);
    }

    // overload operator *
    public static Fraction operator *(Fraction a, Fraction b)
    {
        return new Fraction(a.num * b.num, a.den * b.den);
    }

    // user-defined conversion from Fraction to double
    public static implicit operator double(Fraction f)
    {
        return (double)f.num / f.den;
    }
}

class Test
{
    static void Main()
    {
        Fraction a = new Fraction(1, 2);
        Fraction b = new Fraction(3, 7);
        Fraction c = new Fraction(2, 3);
        Console.WriteLine((double)(a * b + c));
    }
}
/*
Output
0.880952380952381
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

implicit（C# 参考）

explicit（C# 参考）

如何：在结构间实现用户定义的转换（C# 编程指南）

访问关键字（C# 参考）

本节介绍下列访问关键字：

- base

  访问基类的成员。

- this

  引用类的当前实例。

请参阅

C# 参考

C# 编程指南

访问修饰符（C# 参考）

C# 关键字

base（C# 参考）

base 关键字用于从派生类中访问基类的成员：

- 调用基类上已被其他方法重写的方法。
- 指定创建派生类实例时应调用的基类构造函数。

基类访问只能在构造函数、实例方法或实例属性访问器中进行。

从静态方法中使用 **base** 关键字是错误的。

所访问的基类是类声明中指定的基类。例如，如果指定 class ClassB : ClassA，则无论 ClassA 的基类如何，从 ClassB 上访问 ClassA 的成员。

在本例中，基类 Person 和派生类 Employee 都有一个名为 Getinfo 的方法。通过使用 **base** 关键字，可以从派生类中调用基类的 Getinfo 方法。

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}
class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

有关其他示例，请参见 new、virtual 和 override。

本示例显示如何指定在创建派生类实例时调用的基类构造函数。

```
public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    {
    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

this（C# 参考）

this（C# 参考）

this 关键字引用类的当前实例，还可用作扩展方法的第一个参数的修饰符。

> 注意
>
> 本文讨论对类实例使用 **this**。有关其在扩展方法中使用的更多信息，请参见扩展方法（C# 编程指南）。

以下是 **this** 的常用用途：

- 限定被相似的名称隐藏的成员，例如：

```
public Employee(string name, string alias)
{
    // Use this to qualify the fields, name and alias:
    this.name = name;
    this.alias = alias;
}
```

- 将对象作为参数传递到其他方法，例如：

```
CalcTax(this);
```

- 声明索引器，例如：

```
public int this[int param]
{
    get { return array[param]; }
    set { array[param] = value; }
}
```

由于静态成员函数存在于类一级，并且不是对象的一部分，因此没有 **this** 指针。在静态方法中引用 **this** 是错误的。

在本例中，**this** 用于限定 Employee 类成员 name 和 alias，它们都被相似的名称隐藏。该关键字还用于将对象传递到属于其他类的方法 CalcTax。

```
class Employee
{
    private string name;
    private string alias;
    private decimal salary = 3000.00m;
```

```
    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias:
        this.name = name;
        this.alias = alias;
    }
    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
        // Passing the object to the CalcTax method by using this:
        Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }

    public decimal Salary
    {
        get { return salary; }
    }
}

class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return 0.08m * E.Salary;
    }
}

class MainClass
{
    static void Main()
    {
        // Create objects:
        Employee E1 = new Employee("Mingda Pan", "mpan");

        // Display results:
        E1.printEmployee();
    }
}
/*
Output:
    Name: Mingda Pan
    Alias: mpan
    Taxes: $240.00
 */
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

base（C# 参考）

方法（C# 编程指南）

文字关键字（C# 参考）

C# 有下列文字关键字：

- null
- true
- false
- default

请参阅

C# 参考

C# 编程指南

C# 关键字

null（C# 参考）

null 关键字是表示不引用任何对象的 null 引用的文字值。 **null** 是引用类型变量的默认值。普通值类型不能为 null。但是，C# 2.0 引入了可以为 null 值的类型。请参见 可以为 null 的类型（C# 编程指南）。

下面的示例演示 null 关键字的一些行为：

```
class Program
{
    class MyClass
    {
        public void MyMethod() { }
    }

    static void Main(string[] args)
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        mc.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = String.Empty; // Logically the same as ""

        // Equals applied to any null object returns false.
        bool b = (t.Equals(s));
        Console.WriteLine(b);
```

```
        // Equality operator also returns false when one
        // operand is null.
        Console.WriteLine("Empty string {0} null string", s == t ?

        // Returns true.
        Console.WriteLine("null == null is {0}", null == null);

        // A value type cannot be null
        // int i = null; // Compiler error!

        // Use a nullable value type instead:
        int? i = null;

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();

    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

文字关键字（C# 参考）

默认值表（C# 参考）

Nothing (Visual Basic)

default（C# 参考）

default 关键字。

default 关键字可在 **switch** 语句或泛型代码中使用。

- switch 语句：指定默认标签。
- 泛型代码：指定类型参数的默认值。这对于引用类型为 null，对于值类型为零。

请参阅

C# 参考

C# 编程指南

C# 关键字

上下文关键字（C# 参考）

上下文关键字用于提供代码中的特定含义，但它不是 C# 中的保留字。本节介绍下面这些上下文关键字：

关键字	说明
添加	定义一个自定义事件访问器，客户端代码订阅事件时将调用该访问器。
async	指示所修改的方法、 lambda 表达式或匿名方法是异步的。
等待	挂起异步方法，直到其中一等待任务完成。
dynamic	定义一个引用类型，实现发生绕过编译时类型检查的操作。
get	为属性或索引器定义访问器方法。
global	指定未以其他方式命名的默认全局命名空间。
partial — 部分	在整个同一编译单元内定义分部类、结构和接口。
remove	定义一个自定义事件访问器，客户端代码取消订阅事件时将调用该访问器。
set	为属性或索引器定义访问器方法。
value	用于设置访问器和添加或移除事件处理程序。
var	使编译器能够确定在方法作用域中声明的变量的类型。
where	将约束添加到泛型声明。（另请参见 where）。
yield	在迭代器块中使用，用于向枚举数对象返回值或发信号结束迭代。

C# 3.0 中引入的所有查询关键字也都是上下文相关的。有关更多信息，请参见 查询关键字（C# 参考）。

请参阅

C# 参考

C# 编程指南

C# 关键字

add（C# 参考）

add 上下文关键字用于定义一个自定义事件访问器，当客户端代码订阅您的事件时将调用该访问器。如果提供自定义 **add** 访问器，还必须提供 remove 访问器。

下面的示例演示一个具有自定义 **add** 和 remove 访问器的事件。有关完整的示例，请参见如何：实现接口事件（C# 编程指南）。

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add
        {
            lock (PreDrawEvent)
            {
                PreDrawEvent += value;
            }
        }
        remove
        {
            lock (PreDrawEvent)
            {
                PreDrawEvent -= value;
            }
        }
    }

}
```

通常不需要提供自己的自定义事件访问器。在大多数情况下，使用在声明事件时由编译器自动生成的访问器就足够了。

请参阅

事件（C# 编程指南）

get（C# 参考）

get 关键字在属性或索引器中定义访问器方法，以检索该属性或该索引器元素的值。有关更多信息，请参见属性（C# 编程指南）、自动实现的属性（C# 编程指南）和索引器（C# 编程指南）。

下面的示例是名为 Seconds 的属性中的 **get** 访问器：

```
class TimePeriod
{
    private double _seconds;
    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }

}
```

下面的示例是自动实现的属性中的 **get** 访问器：

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

global（C# 参考）

在 :: 运算符前面使用的 **global** 上下文关键字引用全局命名空间，该命名空间是任何 C# 程序的默认命名空间，未以其他方式命名。有关更多信息，请参见 如何：使用全局命名空间别名（C# 编程指南）。

下面的示例演示如何使用 **global** 上下文关键字指定在全局命名空间中定义 TestApp 类：

```
class TestClass : global::TestApp { }
```

请参阅

命名空间（C# 编程指南）

分部（类型）（C# 参考）

分部类型定义允许将类、结构或接口的定义拆分到多个文件中。

在 File1.cs 中：

```
namespace PC
{
    partial class A
    {
        int num = 0;
        void MethodA() { }
        partial void MethodC();
    }
}
```

在 File2.cs 中，声明：

```
namespace PC
{
    partial class A
    {
        void MethodB() { }
        partial void MethodC() { }
    }
}
```

备注

当使用大项目或自动生成的代码（如由 提供的代码）时，将一个类、结构或接口类型拆分到多个文件中的做法就很有用。3c3d61f8-f36c-4d41-b9c3-398376fabb15分部类型可能包含分部方法。有关更多信息，请参见 分部类和方法（C# 编程指南）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

修饰符（C# 参考）

泛型介绍（C# 编程指南）

分部（方法）（C# 参考）

分部方法在分部类型的一个部分中定义它的签名，并在该类型的另外一个部分中定义它的实现。类设计人员可以使用分部方法提供由开发人员决定是否实现的方法挂钩（类似于事件处理程序）。如果开发人员没有提供实现，则编译器会在编译时移除签名。下列条件适用于分部方法：

- 分部类型的两个部分中的签名必须匹配。
- 方法必须返回 void。
- 没有允许的访问修饰符。分部方法是隐式私有的。

下面的示例演示在分部类的两个部分中定义的分部方法：

```
namespace PM
{
    partial class A
    {
        partial void OnSomethingHappened(string s);
    }

    // This part can be in a separate file.
    partial class A
    {
        // Comment out this method and the program
        // will still compile.
        partial void OnSomethingHappened(String s)
        {
            Console.WriteLine("Something happened: {0}", s);
        }
    }
}
```

有关详细信息，请参阅分部类和方法（C# 编程指南）。

请参阅

C# 参考

分部（类型）（C# 参考）

remove（C# 参考）

remove 上下文关键字用于定义一个自定义事件访问器，当客户端代码取消订阅事件时将调用该访问器。 如果提供自定义 remove 访问器，还必须提供 add 访问器。

示例

下面的示例演示一个具有自定义 add 和 remove 访问器的事件。 有关完整的示例，请参见如何：实现接口事件（C# 编程指南）。

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add
        {
            lock (PreDrawEvent)
            {
                PreDrawEvent += value;
            }
        }
        remove
        {
            lock (PreDrawEvent)
            {
                PreDrawEvent -= value;
            }
        }
    }

}
```

通常不需要提供自己的自定义事件访问器。 在大多数情况下，使用在声明事件时由编译器自动生成的访问器就足够了。

请参见

参考

事件（C# 编程指南）

set（C# 参考）

set 关键字在属性或索引器中定义为该属性或索引器元素赋值的“访问器”方法。有关详细信息和示例，请参阅属性（C# 编程指南），自动实现的属性（C# 编程指南），和索引器（C# 编程指南）。

此例是名为 Seconds 的属性的 **set** 访问器：

```
class TimePeriod
{
    private double _seconds;
    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }

}
```

下面的示例是自动实现的属性中的 **set** 访问器：

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

属性（C# 编程指南）

where（泛型类型约束）（C# 参考）

在泛型类型定义中，**where** 子句用于指定对下列类型的约束：这些类型可用作泛型声明中定义的类型参数的实参。 例如，可以声明一个泛型类 MyGenericClass，这样，类型参数 T 就可以实现 IComparable<T> 接口：

```
public class MyGenericClass<T> where T:IComparable { }
```

> 注意
>
> 有关查询表达式中的 where 子句的更多信息，请参见 where 子句（C# 参考）。

除了接口约束，**where** 子句还可以包括基类约束，以指出某个类型必须将指定的类作为基类（或者就是该类本身），才能用作该泛型类型的类型参数。 这样的约束一经使用，就必须出现在该类型参数的所有其他约束之前。

```
class MyClass<T, U>
    where T : class
    where U : struct
{ }
```

where 子句还可以包括构造函数约束。 可以使用 new 运算符创建类型参数的实例；但类型参数为此必须受构造函数约束 **new()** 的约束。 new() 约束可以让编译器知道：提供的任何类型参数都必须具有可访问的无参数（或默认）构造函数。 例如：

```
public class MyGenericClass<T> where T : IComparable, new()
{
    // The following line is not possible without new() constraint
    T item = new T();
}
```

new() 约束出现在 **where** 子句的最后。

对于多个类型参数，每个类型参数都使用一个 **where** 子句，例如：

```
interface IMyInterface
{
}

class Dictionary<TKey, TVal>
    where TKey : IComparable, IEnumerable
    where TVal : IMyInterface
{
    public void Add(TKey key, TVal val)
    {
    }
}
```

还可以将约束附加到泛型方法的类型参数，例如：

```
public bool MyMethod<T>(T t) where T : IMyInterface { }
```

请注意，对于委托和方法两者来说，描述类型参数约束的语法是一样的：

```
delegate T MyDelegate<T>() where T : new()
```

有关泛型委托的信息，请参见泛型委托。

有关约束的语法和用法的详细信息，请参见类型参数的约束。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

泛型介绍（C# 编程指南）

new 约束（C# 参考）

类型参数的约束（C# 编程指南）

value（C# 参考）

上下文关键字 **value** 用在普通属性声明的 set 访问器中。此关键字类似于方法的输入参数。关键字 **value** 引用客户端代码尝试赋给属性的值。在下面的示例中，MyDerivedClass 有一个名为 Name 的属性，该属性使用 **value** 参数向支持字段 name 分配新字符串。从客户端代码的角度来看，该操作写作一个简单的赋值语句。

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set a
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string name;

   // Override auto-implemented property with ordinary property
   // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (value != String.Empty)
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }

}
```

有关使用 **value** 的更多信息，请参见属性（C# 编程指南）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 关键字

yield（C# 参考）

如果你在语句中使用 **yield** 关键字，则意味着它在其中出现的方法、运算符或 **get** 访问器是迭代器。通过使用 **yield** 定义迭代器，可在实现自定义集合类型的 IEnumerable 和 IEnumerator 模式时无需其他显式类（保留枚举状态的类，有关示例，请参阅 IEnumerator<T>）。

下面的示例演示了 **yield** 语句的两种形式。

```
yield return <expression>;
yield break;
```

备注

使用 **yield return** 语句可一次返回一个元素。

通过 foreach 语句或 LINQ 查询来使用迭代器方法。 **foreach** 循环的每次迭代都会调用迭代器方法。迭代器方法运行到 **yield return** 语句时，会返回一个 *expression*，并保留当前在代码中的位置。 当下次调用迭代器函数时执行从该位置重新启动。

可以使用 **yield break** 语句来终止迭代。

有关迭代器的详细信息，请参阅迭代器（C# 和 Visual Basic）。

迭代器方法和 get 访问器

迭代器的声明必须满足以下要求：

- 返回类型必须为 IEnumerable、IEnumerable<T>、IEnumerator 或 IEnumerator<T>。
- 该声明不能有任何 ref 或out https://msdn.microsoft.com/zh-CN/library/t3c3bfhx.aspx 参数。

返回 IEnumerable 或 IEnumerator 的迭代器的 **yield** 类型为 **object**。如果迭代器返回 IEnumerable<T> 或 IEnumerator<T>，则必须将 **yield return** 语句中的表达式类型隐式转换为泛型类型参数。

你不能在具有以下特点的方法中包含 **yield return** 或 **yield break** 语句：

- 匿名方法。有关详细信息，请参阅匿名方法（C# 编程指南）。
- 包含不安全的块的方法。有关详细信息，请参阅unsafe（C# 参考）。

异常处理

不能将 **yield return** 语句置于 try-catch 块中。可将 **yield return** 语句置于 try-finally 语句的 try 块中。

yield break 语句可以位于 try 块或 catch 块，但不能位于 finally 块。

如果 **foreach** 主体（在迭代器方法之外）引发异常，则将执行迭代器方法中的 **finally** 块。

技术实现

以下代码从迭代器方法返回 **IEnumerable<string>**，然后遍历其元素。

```
IEnumerable<string> elements = MyIteratorMethod();
foreach (string element in elements)
{
   …
}
```

调用 MyIteratorMethod 并不执行该方法的主体。相反，该调用会将 **IEnumerable<string>** 返回到 elements 变量中。

在 **foreach** 循环迭代时，将为 elements 调用 MoveNext 方法。此调用将执行 MyIteratorMethod 的主体，直至到达下一个 **yield return** 语句。 **yield return** 语句返回的表达式不仅决定了循环体使用的 element 变量值，还决定了元素的 Current 属性（它是 **IEnumerable<string>**）。

在 **foreach** 循环的每个后续迭代中，迭代器主体的执行将从它暂停的位置继续，直至到达 **yield return** 语句后才会停止。在到达迭代器方法的结尾或 **yield break** 语句时，**foreach** 循环便已完成。

下面的示例包含一个位于 **for** 循环内的 **yield return** 语句。 Process 中的 **foreach** 语句体的每次迭代都会创建对 Power 迭代器函数的调用。对迭代器函数的每个调用将继续到 **yield return** 语句的下一次执行（在 **for** 循环的下一次迭代期间发生）。

迭代器方法的返回类型是 IEnumerable（一种迭代器接口类型）。当调用迭代器方法时，它将返回一个包含数字幂的可枚举对象。

```
public class PowersOf2
{
    static void Main()
    {
        // Display powers of 2 up to the exponent of 8:
        foreach (int i in Power(2, 8))
        {
            Console.Write("{0} ", i);
        }
    }

    public static System.Collections.Generic.IEnumerable<int> Power
    {
        int result = 1;

        for (int i = 0; i < exponent; i++)
        {
            result = result * number;
            yield return result;
        }
    }

    // Output: 2 4 8 16 32 64 128 256
}
```

下面的示例演示一个作为迭代器的 **get** 访问器。在该示例中，每个 **yield return** 语句返回一个用户定义的类的实例。

```
public static class GalaxyClass
{
    public static void ShowGalaxies()
    {
        var theGalaxies = new Galaxies();
        foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy)
        {
            Debug.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLi
        }
    }

    public class Galaxies
    {

        public System.Collections.Generic.IEnumerable<Galaxy> NextG
        {
            get
            {
                yield return new Galaxy { Name = "Tadpole", MegaLig
                yield return new Galaxy { Name = "Pinwheel", MegaLi
                yield return new Galaxy { Name = "Milky Way", MegaL
                yield return new Galaxy { Name = "Andromeda", MegaL
            }
        }

    }

    public class Galaxy
    {
        public String Name { get; set; }
        public int MegaLightYears { get; set; }
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

foreach，in（C# 参考）

迭代器（C# 和 Visual Basic）

查询关键字（C# 参考）

本节包含查询表达式中使用的上下文关键字。

本节内容

子句	说明
from	指定数据源和范围变量（类似于迭代变量）。
where	根据一个或多个由逻辑“与”和逻辑“或”运算符（&& 或 ||）分隔的布尔表达式筛选源元素。
select	指定当执行查询时返回的序列中的元素将具有的类型和形式。
group	按照指定的键值对查询结果进行分组。
into	提供一个标识符，它可以充当对 join、group 或 select 子句的结果的引用。
orderby	基于元素类型的默认比较器按升序或降序对查询结果进行排序。
join	基于两个指定匹配条件之间的相等比较来联接两个数据源。
let	引入一个用于存储查询表达式中的子表达式结果的范围变量。
in	join 子句中的上下文关键字。
on	join 子句中的上下文关键字。
equals	join 子句中的上下文关键字。
by	group 子句中的上下文关键字。
ascending	orderby 子句中的上下文关键字。
descending	orderby 子句中的上下文关键字。

请参阅

C# 关键字

LINQ (Language-Integrated Query)

LINQ 查询表达式（C# 编程指南）

Getting Started with LINQ in C#

from 子句（C# 参考）

查询表达式必须以 **from** 子句开头。另外，查询表达式还可以包含子查询，子查询也是以 **from** 子句开头。 **from** 子句指定以下内容：

- 将对其运行查询或子查询的数据源。
- 一个本地范围变量，表示源序列中的每个元素。

范围变量和数据源都是强类型。 **from** 子句中引用的数据源的类型必须为 IEnumerable、IEnumerable<T> 或一种派生类型（如 IQueryable<T>）。

在下面的示例中，numbers 是数据源，而 num 是范围变量。请注意，这两个变量都是强类型，即使使用了 var 关键字也是如此。

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 4 1 3 2 0
```

范围变量

如果数据源实现了 IEnumerable<T>，则编译器可以推断范围变量的类型。例如，如果数据源的类型为 **IEnumerable<Customer>**，则推断出范围变量的类型为 Customer。仅当数据源是非泛型 **IEnumerable** 类型（如 ArrayList）时，才必须显式指定数据源类型。有关更多信息，请参见 How to: Query an ArrayList with LINQ。

在上一个示例中，num 被推断为 **int** 类型。由于范围变量是强类型，因此可以对其调用方法或者在其他操作中使用它。例如，可以不编写 select num，而编写 select num.ToString() 使查询表达式返回一个字符串序列而不是整数序列。或者，也可以编写 select n + 10 使表达式返回序列 14、11、13、12、10。有关更多信息，请参见 select 子句（C# 参考）。

范围变量类似于 foreach 语句中的迭代变量，只是两者之间有一个非常重要的区别：范围变量从不实际存储来自数据源的数据。范围变量只是提供了语法上的便利，使查询能够描述执行查询时将发生的事情。有关更多信息，请参见 Introduction to LINQ Queries (C#)。

复合 from 子句

在某些情况下，源序列中的每个元素本身可能是序列，也可能包含序列。例如，数据源可能是一个 **IEnumerable<Student>**，其中，序列中的每个 Student 对象都包含一个测验得分列表。若要访问每个 Student 元素中的内部列表，可以使用复合 **from** 子句。该技术类似于使用嵌套的 foreach 语句。可以向任一 **from** 子句中添加 where 或 orderby 子句来筛选结果。下面的示例演示了一个 Student 对象序列，其中每个对象都包含一个表示测验得分的内部整数 List。为了访问该内部列表，此示例使用了复合 **from** 子句。如有必要，可在两个 **from** 子句之间再插入子句。

```
class CompoundFrom
{
    // The element type of the data source.
    public class Student
    {
        public string LastName { get; set; }
        public List<int> Scores {get; set;}
    }

    static void Main()
    {

        // Use a collection initializer to create the data source.
        // each element in the list contains an inner sequence of s
        List<Student> students = new List<Student>
        {
           new Student {LastName="Omelchenko", Scores= new List<int
           new Student {LastName="O'Donnell", Scores= new List<int>
           new Student {LastName="Mortensen", Scores= new List<int>
           new Student {LastName="Garcia", Scores= new List<int> {9
           new Student {LastName="Beebe", Scores= new List<int> {35
        };

        // Use a compound from to access the inner sequence within
        // Note the similarity to a nested foreach statement.
        var scoreQuery = from student in students
                         from score in student.Scores
                            where score > 90
```

```
                              select new { Last = student.LastName, s

        // Execute the queries.
        Console.WriteLine("scoreQuery:");
        // Rest the mouse pointer on scoreQuery in the following li
        // see its type. The type is IEnumerable<'a>, where 'a is a
        // anonymous type defined as new {string Last, int score}.
        // each instance of this anonymous type has two members, a
        // (Last) and an int (score).
        foreach (var student in scoreQuery)
        {
            Console.WriteLine("{0} Score: {1}", student.Last, stude
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
Beebe Score: 91
*/
```

使用多个 from 子句执行联接

复合 **from** 子句用于访问单个数据源中的内部集合。不过，查询还可以包含多个可从独立数据源生成补充查询的 **from** 子句。使用此技术可以执行某些类型的、无法通过使用 join 子句执行的联接操作。

下面的示例演示如何使用两个 **from** 子句构成两个数据源的完全交叉联接。

```
class CompoundFrom2
{
    static void Main()
    {
        char[] upperCase = { 'A', 'B', 'C' };
        char[] lowerCase = { 'x', 'y', 'z' };

        // The type of joinQuery1 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery1 =
            from upper in upperCase
            from lower in lowerCase
```

```
            select new { upper, lower };

        // The type of joinQuery2 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery2 =
            from lower in lowerCase
            where lower != 'x'
            from upper in upperCase
            select new { lower, upper };

        // Execute the queries.
        Console.WriteLine("Cross join:");
        // Rest the mouse pointer on joinQuery1 to verify its type.
        foreach (var pair in joinQuery1)
        {
            Console.WriteLine("{0} is matched to {1}", pair.upper,
        }

        Console.WriteLine("Filtered non-equijoin:");
        // Rest the mouse pointer over joinQuery2 to verify its typ
        foreach (var pair in joinQuery2)
        {
            Console.WriteLine("{0} is matched to {1}", pair.lower,
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
        Cross join:
        A is matched to x
        A is matched to y
        A is matched to z
        B is matched to x
        B is matched to y
        B is matched to z
        C is matched to x
        C is matched to y
        C is matched to z
        Filtered non-equijoin:
        y is matched to A
        y is matched to B
        y is matched to C
        z is matched to A
        z is matched to B
        z is matched to C
        */
```

有关使用多个 **from** 子句的联接操作的更多信息，请参见如何：执行自定义联接操作（C# 编程指南）。

请参阅

查询关键字（C# 参考）

LINQ 查询表达式（C# 编程指南）

where 子句（C# 参考）

where 子句用在查询表达式中，用于指定将在查询表达式中返回数据源中的哪些元素。它将一个布尔条件（“谓词”）应用于每个源元素（由范围变量引用），并返回满足指定条件的元素。一个查询表达式可以包含多个 **where** 子句，一个子句可以包含多个谓词子表达式。

在下面的示例中，**where** 子句筛选出除小于五的数字外的所有数字。如果移除 **where** 子句，则会返回数据源中的所有数字。表达式 num < 5 是应用于每个元素的谓词。

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

在单一 **where** 子句内，可以使用 && 和 || 运算符根据需要指定任意多个谓词。在下面的示例中，查询将指定两个谓词，以便只选择小于五的偶数。

```
class WhereSample2
{
    static void Main()
    {
        // Data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with two predicates in where clause.
        var queryLowNums2 =
            from num in numbers
            where num < 5 && num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums2)
        {
            Console.Write(s.ToString() + " ");
        }
        Console.WriteLine();

        // Create the query with two where clause.
        var queryLowNums3 =
            from num in numbers
            where num < 5
            where num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums3)
        {
            Console.Write(s.ToString() + " ");
        }

    }
}
// Output:
// 4 2 0
// 4 2 0
```

where 子句可以包含一个或多个返回布尔值的方法。在下面的示例中，**where** 子句使用一个方法来确定范围变量的当前值是偶数还是奇数。

```
class WhereSample3
{
    static void Main()
    {
        // Data source
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

         // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }

    // Method may be instance method or static method.
    static bool IsEven(int i)
    {
        return i % 2 == 0;
    }
}
//Output: 4 8 6 2 0
```

备注

where 子句是一种筛选机制。除了不能是第一个或最后一个子句外，它几乎可以放在查询表达式中的任何位置。 **where** 子句可以出现在 group 子句的前面或后面，具体情况取决于是必须在对源元素进行分组之前还是之后来筛选源元素。

如果指定的谓词对于数据源中的元素无效，则会发生编译时错误。这是 LINQ 提供的强类型检查的一个优点。

编译时，**where** 关键字会被转换为对 Where 标准查询运算符方法的调用。

请参阅

查询关键字（C# 参考）

from 子句（C# 参考）

select 子句（C# 参考）

Filtering Data

LINQ 查询表达式（C# 编程指南）

Getting Started with LINQ in C#

select 子句（C# 参考）

在查询表达式中，**select** 子句可以指定将在执行查询时产生的值的类型。该子句的结果将基于前面所有子句的计算结果以及 **select** 子句本身中的所有表达式。查询表达式必须以 **select** 子句或 group 子句结束。

下面的示例演示了查询表达式中的简单 **select** 子句。

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = new List<int>() { 97, 92, 81, 60 };

        // Create the query.
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.Write(i + " ");
        }
    }
}
//Output: 97 92 81
```

select 子句产生的序列的类型决定了查询变量 queryHighScores 的类型。在最简单的情况下，**select** 子句仅指定范围变量。这会使返回的序列包含与数据源具有相同类型的元素。有关更多信息，请参见Type Relationships in LINQ Query Operations (C#)。不过，**select** 子句还提供了一种功能强大的机制，可用于将源数据转换（或投影）为新类型。有关更多信息，请参见 Data Transformations with LINQ (C#)。

下面的示例演示了 **select** 子句可能采用的所有不同形式。在每个查询中，请注意 **select** 子句和查询变量（studentQuery1、studentQuery2 等）的类型之间的关系。

```
    class SelectSample2
    {
        // Define some classes
        public class Student
        {
            public string First { get; set; }
            public string Last { get; set; }
```

```
            public int ID { get; set; }
            public List<int> Scores;
            public ContactInfo GetContactInfo(SelectSample2 app, in
            {
                ContactInfo cInfo =
                    (from ci in app.contactList
                    where ci.ID == id
                    select ci)
                    .FirstOrDefault();

                return cInfo;
            }

            public override string ToString()
            {
                return First + " " + Last + ":" + ID;
            }
        }

        public class ContactInfo
        {
            public int ID { get; set; }
            public string Email { get; set; }
            public string Phone { get; set; }
            public override string ToString() { return Email + ","
        }

        public class ScoreInfo
        {
            public double Average { get; set; }
            public int ID { get; set; }
        }

        // The primary data source
        List<Student> students = new List<Student>()
        {
             new Student {First="Svetlana", Last="Omelchenko", ID=1
             new Student {First="Claire", Last="O'Donnell", ID=112,
             new Student {First="Sven", Last="Mortensen", ID=113, S
             new Student {First="Cesar", Last="Garcia", ID=114, Sco
        };

        // Separate data source for contact info.
        List<ContactInfo> contactList = new List<ContactInfo>()
        {
            new ContactInfo {ID=111, Email="SvetlanO@Contoso.com",
            new ContactInfo {ID=112, Email="ClaireO@Contoso.com", P
            new ContactInfo {ID=113, Email="SvenMort@Contoso.com",
            new ContactInfo {ID=114, Email="CesarGar@Contoso.com",
        };

        static void Main(string[] args)
        {
```

```
            SelectSample2 app = new SelectSample2();

            // Produce a filtered sequence of unmodified Students.
            IEnumerable<Student> studentQuery1 =
                from student in app.students
                where student.ID > 111
                select student;

            Console.WriteLine("Query1: select range_variable");
            foreach (Student s in studentQuery1)
            {
                Console.WriteLine(s.ToString());
            }

            // Produce a filtered sequence of elements that contair
            // only one property of each Student.
            IEnumerable<String> studentQuery2 =
                from student in app.students
                where student.ID > 111
                select student.Last;

            Console.WriteLine("\r\n studentQuery2: select range_var
            foreach (string s in studentQuery2)
            {
                Console.WriteLine(s);
            }

            // Produce a filtered sequence of objects created by
            // a method call on each Student.
            IEnumerable<ContactInfo> studentQuery3 =
                from student in app.students
                where student.ID > 111
                select student.GetContactInfo(app, student.ID);

            Console.WriteLine("\r\n studentQuery3: select range_var
            foreach (ContactInfo ci in studentQuery3)
            {
                Console.WriteLine(ci.ToString());
            }

            // Produce a filtered sequence of ints from
            // the internal array inside each Student.
            IEnumerable<int> studentQuery4 =
                from student in app.students
                where student.ID > 111
                select student.Scores[0];

            Console.WriteLine("\r\n studentQuery4: select range_var
            foreach (int i in studentQuery4)
            {
                Console.WriteLine("First score = {0}", i);
            }
```

```
            // Produce a filtered sequence of doubles
            // that are the result of an expression.
            IEnumerable<double> studentQuery5 =
                from student in app.students
                where student.ID > 111
                select student.Scores[0] * 1.1;

            Console.WriteLine("\r\n studentQuery5: select expressio
            foreach (double d in studentQuery5)
            {
                Console.WriteLine("Adjusted first score = {0}", d);
            }

            // Produce a filtered sequence of doubles that are
            // the result of a method call.
            IEnumerable<double> studentQuery6 =
                from student in app.students
                where student.ID > 111
                select student.Scores.Average();

            Console.WriteLine("\r\n studentQuery6: select expressio
            foreach (double d in studentQuery6)
            {
                Console.WriteLine("Average = {0}", d);
            }

            // Produce a filtered sequence of anonymous types
            // that contain only two properties from each Student.
            var studentQuery7 =
                from student in app.students
                where student.ID > 111
                select new { student.First, student.Last };

            Console.WriteLine("\r\n studentQuery7: select new anony
            foreach (var item in studentQuery7)
            {
                Console.WriteLine("{0}, {1}", item.Last, item.First
            }

            // Produce a filtered sequence of named objects that co
            // a method return value and a property from each Stude
            // Use named types if you need to pass the query variab
            // across a method boundary.
            IEnumerable<ScoreInfo> studentQuery8 =
                from student in app.students
                where student.ID > 111
                select new ScoreInfo
                {
                    Average = student.Scores.Average(),
                    ID = student.ID
                };

            Console.WriteLine("\r\n studentQuery8: select new named
```

```
            foreach (ScoreInfo si in studentQuery8)
            {
                Console.WriteLine("ID = {0}, Average = {1}", si.ID,
            }

            // Produce a filtered sequence of students who appear c
            // and whose average is greater than 85.
            IEnumerable<ContactInfo> studentQuery9 =
                from student in app.students
                where student.Scores.Average() > 85
                join ci in app.contactList on student.ID equals ci.
                select ci;

            Console.WriteLine("\r\n studentQuery9: select result of
            foreach (ContactInfo ci in studentQuery9)
            {
                Console.WriteLine("ID = {0}, Email = {1}", ci.ID, c
            }

            // Keep the console window open in debug mode
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
            }
        }
    /* Output
        Query1: select range_variable
        Claire O'Donnell:112
        Sven Mortensen:113
        Cesar Garcia:114

        studentQuery2: select range_variable.Property
        O'Donnell
        Mortensen
        Garcia

        studentQuery3: select range_variable.Method
        ClaireO@Contoso.com,206-555-0298
        SvenMort@Contoso.com,206-555-1130
        CesarGar@Contoso.com,206-555-0521

        studentQuery4: select range_variable[index]
        First score = 75
        First score = 88
        First score = 97

        studentQuery5: select expression
        Adjusted first score = 82.5
        Adjusted first score = 96.8
        Adjusted first score = 106.7

        studentQuery6: select expression2
        Average = 72.25
        Average = 84.5
```

```
        Average = 88.25

        studentQuery7: select new anonymous type
        O'Donnell, Claire
        Mortensen, Sven
        Garcia, Cesar

        studentQuery8: select new named type
        ID = 112, Average = 72.25
        ID = 113, Average = 84.5
        ID = 114, Average = 88.25

        studentQuery9: select result of join clause
        ID = 114, Email = CesarGar@Contoso.com
*/
```

如上一个示例中的 studentQuery8 所示，您有时可能希望所返回序列中的元素仅包含源元素的属性子集。通过使返回的序列尽可能地小一些，可以降低内存需求，并提高查询的执行速度。通过在 **select** 子句中创建一个匿名类型，并且借助于对象初始值设定项用源元素中的适当属性对该匿名类型进行初始化，可以达到此目的。有关如何执行此操作的示例，请参见对象和集合初始值设定项（C# 编程指南）。

备注

编译时，**select** 子句会被转换为对 Select<TSource, TResult> 标准查询运算符的方法调用。

请参阅

C# 参考

查询关键字（C# 参考）

from 子句（C# 参考）

分部（方法）（C# 参考）

匿名类型（C# 编程指南）

LINQ 查询表达式（C# 编程指南）

Getting Started with LINQ in C#

group 子句（C# 参考）

group 子句返回一个 IGrouping<TKey, TElement> 对象序列，这些对象包含零个或更多个与该组的键值匹配的项。例如，可以按照每个字符串中的第一个字母对字符串序列进行分组。在这种情况下，第一个字母是键且具有 char 类型，并且存储在每个 IGrouping<TKey, TElement> 对象的 **Key** 属性中。编译器可推断该键的类型。

可以用 **group** 子句结束查询表达式，如下面的示例所示：

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

如果您想要对每个组执行附加查询操作，则可以使用 into 上下文关键字指定一个临时标识符。使用 **into** 时，必须继续编写该查询，并最终用一个 **select** 语句或另一个 **group** 子句结束该查询，如下面的代码摘录所示：

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

本主题中的“示例”部分中提供了使用含有和不含 **into** 的 **group** 的更完整示例。

枚举组查询的结果

由于 **group** 查询产生的 IGrouping<TKey, TElement> 对象实质上是列表的列表，因此必须使用嵌套的 foreach 循环来访问每一组中的各个项。外部循环用于循环访问组键，内部循环用于循环访问组本身中的每个项。组可能具有键，但没有元素。以下是执行上述代码示例中的查询的 **foreach** 循环：

```
// Iterate group items with a nested foreach. This IGrouping encaps
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
     Console.WriteLine(studentGroup.Key);
     // Explicit type for student could also be used here.
     foreach (var student in studentGroup)
     {
         Console.WriteLine("   {0}, {1}", student.Last, student.Fi
     }
 }
```

键类型

组键可以是任何类型，如字符串、内置数值类型、用户定义的命名类型或匿名类型。

按字符串进行分组

上述代码示例使用的是 **char**。可以很容易地改为指定字符串键，如完整的姓氏：

```
// Same as previous example except we use the entire last name as a
// Query variable is an IEnumerable<IGrouping<string, Student>>
 var studentQuery3 =
     from student in students
     group student by student.Last;
```

按布尔进行分组

下面的示例演示使用布尔值作为键将结果划分成两个组。请注意，该值是由 **group** 子句中的子表达式产生的。

```
class GroupSample1
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }
```

```
    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source.
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111
           new Student {First="Claire", Last="O'Donnell", ID=112, S
           new Student {First="Sven", Last="Mortensen", ID=113, Sco
           new Student {First="Cesar", Last="Garcia", ID=114, Score
           new Student {First="Debra", Last="Garcia", ID=115, Score
        };

        return students;

    }

    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Group by true or false.
        // Query variable is an IEnumerable<IGrouping<bool, Student
        var booleanGroupQuery =
            from student in students
            group student by student.Scores.Average() >= 80; //pass

        // Execute the query and access items in each group
        foreach (var studentGroup in booleanGroupQuery)
        {
            Console.WriteLine(studentGroup.Key == true ? "High aver
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last,
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
  Low averages
   Omelchenko, Svetlana:77.5
   O'Donnell, Claire:72.25
   Garcia, Cesar:75.5
  High averages
   Mortensen, Sven:93.5
   Garcia, Debra:88.25
*/
```

按数值范围进行分组

下一个示例使用表达式创建表示百分比范围的数值组键。请注意，该示例使用 let 作为方法调用结果的方便存储位置，从而无需在 **group** 子句中调用该方法两次。另请注意，在 **group** 子句中，为了避免发生“被零除”异常，代码进行了相应检查以确保学生的平均成绩不为零。有关如何在查询表达式中安全使用方法的更多信息，请参见如何：在查询表达式中处理异常（C# 编程指南）。

```
class GroupSample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source.
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111
           new Student {First="Claire", Last="O'Donnell", ID=112, S
           new Student {First="Sven", Last="Mortensen", ID=113, Sco
           new Student {First="Cesar", Last="Garcia", ID=114, Score
           new Student {First="Debra", Last="Garcia", ID=115, Score
        };

        return students;

    }

    // This method groups students into percentile ranges based on
    // grade average. The Average method returns a double, so to pr
    // number it is necessary to cast to int before dividing by 10
    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Write the query.
        var studentQuery =
            from student in students
            let avg = (int)student.Scores.Average()
            group student by (avg == 0 ? 0 : avg / 10) into g
            orderby g.Key
```

```
            select g;

        // Execute the query.
        foreach (var studentGroup in studentQuery)
        {
            int temp = studentGroup.Key * 10;
            Console.WriteLine("Students with an average between {0]
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last,
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
     Students with an average between 70 and 80
       Omelchenko, Svetlana:77.5
       O'Donnell, Claire:72.25
       Garcia, Cesar:75.5
     Students with an average between 80 and 90
       Garcia, Debra:88.25
     Students with an average between 90 and 100
       Mortensen, Sven:93.5
 */
```

按复合键进行分组

当您想要按照多个键对元素进行分组时，可使用复合键。通过使用匿名类型或命名类型来存储键元素，可以创建复合键。在下面的示例中，假定已经使用名为 surname 和 city 的两个成员声明了类 Person。 **group** 子句使得为每组具有相同姓氏和相同城市的人员创建一个单独的组。

```
group person by new {name = person.surname, city = person.city};
```

如果必须将查询变量传递给其他方法，请使用命名类型。使用自动实现的属性作为键来创建一个特殊类，然后重写 Equals 和 GetHashCode 方法。还可以使用结构；在此情况下，并不绝对需要重写这些方法。有关更多信息，请参见如何：使用自动实现的属性实现轻量类（C# 编程指南）和How to: Query for Duplicate Files in a Directory Tree (LINQ)。后一个主题包含一个代码示例，该示例演示如何将复合键与命名类型结合使用。

下面的示例演示在没有向组应用附加查询逻辑时将源数据排序放入不同组中的标准模式。这称为不带延续的分组。字符串数组中的元素按照它们的第一个字母进行分组。查询结果是一个 IGrouping<TKey, TElement> 类型，其中包含一个 **char** 类型

的公共 Key 属性以及一个包含分组中每个项的 IEnumerable<T> 集合。

group 子句的结果是序列的序列。因此，若要访问所返回的每个组中的单个元素，请在循环访问组键的循环内使用嵌套的 **foreach** 循环，如下面的示例所示。

```
class GroupExample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = { "blueberry", "chimpanzee", "abacus", "ba

        // Create the query.
        var wordGroups =
            from w in words
            group w by w[0];

        // Execute the query.
        foreach (var wordGroup in wordGroups)
        {
            Console.WriteLine("Words that start with the letter '{0
            foreach (var word in wordGroup)
            {
                Console.WriteLine(word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
      Words that start with the letter 'b':
        blueberry
        banana
      Words that start with the letter 'c':
        chimpanzee
        cheese
      Words that start with the letter 'a':
        abacus
        apple
     */
```

此示例演示在创建组之后，如何使用通过 **into** 实现的延续对这些组执行附加逻辑。有关更多信息，请参见 into（C# 参考）。下面的示例查询每个组以仅选择那些键值为元音的元素。

```
class GroupClauseExample2
{
    static void Main()
    {
        // Create the data source.
        string[] words2 = { "blueberry", "chimpanzee", "abacus", "b

        // Create the query.
        var wordGroups2 =
            from w in words2
            group w by w[0] into grps
            where (grps.Key == 'a' || grps.Key == 'e' || grps.Key =
                   || grps.Key == 'o' || grps.Key == 'u')
            select grps;

        // Execute the query.
        foreach (var wordGroup in wordGroups2)
        {
            Console.WriteLine("Groups that start with a vowel: {0}'
            foreach (var word in wordGroup)
            {
                Console.WriteLine("   {0}", word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Groups that start with a vowel: a
        abacus
        apple
        anteater
    Groups that start with a vowel: e
        elephant
    Groups that start with a vowel: u
        umbrella
*/
```

备注

编译时，**group** 子句被转换为对 GroupBy<TSource, TKey> 方法的调用。

请参阅

IGrouping<TKey，TElement>

GroupBy<TSource，TKey>

ThenBy<TSource，TKey>

ThenByDescending<TSource，TKey>

查询关键字（C# 参考）

LINQ 查询表达式（C# 编程指南）

如何：创建嵌套组（C# 编程指南）

如何：对查询结果进行分组（C# 编程指南）

如何：对分组操作执行子查询（C# 编程指南）

into（C# 参考）

可以使用 **into** 上下文关键字创建一个临时标识符，以便将 group、join 或 select 子句的结果存储到新的标识符中。此标识符本身可以是附加查询命令的生成器。在 **group** 或 **select** 子句中使用新标识符的用法有时称为“延续”。

下面的示例演示使用 **into** 关键字来启用临时标识符 fruitGroup，该标识符具有推断类型 **IGrouping**。通过使用该标识符，可以对每个组调用 Count<TSource> 方法，并且仅选择那些包含两个或更多个单词的组。

```
class IntoSample1
{
    static void Main()
    {

        // Create a data source.
        string[] words = { "apples", "blueberries", "oranges", "bar

        // Create the query.
        var wordGroups1 =
            from w in words
            group w by w[0] into fruitGroup
            where fruitGroup.Count() >= 2
            select new { FirstLetter = fruitGroup.Key, Words = frui

        // Execute the query. Note that we only iterate over the gr
        // not the items in each group
        foreach (var item in wordGroups1)
        {
            Console.WriteLine(" {0} has {1} elements.", item.FirstL
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   a has 2 elements.
   b has 2 elements.
*/
```

仅当希望对每个组执行附加查询操作时，才需要在 **group** 子句中使用 **into**。有关更多信息，请参见 group 子句（C# 参考）。

有关在 **join** 子句中使用 **into** 的示例，请参见 join 子句（C# 参考）。

请参阅

查询关键字（C# 参考）

LINQ 查询表达式（C# 编程指南）

group 子句（C# 参考）

orderby 子句（C# 参考）

在查询表达式中，**orderby** 子句可使返回的序列或子序列（组）按升序或降序排序。可以指定多个键，以便执行一个或多个次要排序操作。排序是由针对元素类型的默认比较器执行的。默认排序顺序为升序。您还可以指定自定义比较器。但是，只能通过基于方法的语法使用它。有关更多信息，请参见 Sorting Data。

在下面的示例中，第一个查询按从 A 开始的字母顺序对单词进行排序，第二个查询按降序对相同的单词进行排序。（**ascending** 关键字是默认排序值，可以省略。）

```
class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = { "cherry", "apple", "blueberry" };

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit //"ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "Descending:");
        foreach (string s in sortDescendingQuery)
        {
            Console.WriteLine(s);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/
```

下面的示例对学生的姓氏执行主要排序，然后对他们的名字执行次要排序。

```
class OrderbySample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source.
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111
           new Student {First="Claire", Last="O'Donnell", ID=112},
           new Student {First="Sven", Last="Mortensen", ID=113},
           new Student {First="Cesar", Last="Garcia", ID=114},
           new Student {First="Debra", Last="Garcia", ID=115}
        };

        return students;

    }
    static void Main(string[] args)
    {
        // Create the data source.
        List<Student> students = GetStudents();

        // Create the query.
        IEnumerable<Student> sortedStudents =
            from student in students
            orderby student.Last ascending, student.First ascending
            select student;

        // Execute the query.
        Console.WriteLine("sortedStudents:");
        foreach (Student student in sortedStudents)
            Console.WriteLine(student.Last + " " + student.First);

        // Now create groups and sort the groups. The query first s
        // of all students so that they will be in alphabetical ord
        // grouped. The second orderby sorts the group keys in alph
        var sortedGroups =
            from student in students
            orderby student.Last, student.First
            group student by student.Last[0] into newGroup
            orderby newGroup.Key
            select newGroup;
```

```
        // Execute the query.
        Console.WriteLine(Environment.NewLine + "sortedGroups:");
        foreach (var studentGroup in sortedGroups)
        {
            Console.WriteLine(studentGroup.Key);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}", student.Last, stuc
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
sortedStudents:
Garcia Cesar
Garcia Debra
Mortensen Sven
O'Donnell Claire
Omelchenko Svetlana

sortedGroups:
G
   Garcia, Cesar
   Garcia, Debra
M
   Mortensen, Sven
O
   O'Donnell, Claire
   Omelchenko, Svetlana
*/
```

备注

编译时，**orderby** 子句被转换为对 OrderBy<TSource, TKey> 方法的调用。**orderby** 子句中的多个键转换为 ThenBy<TSource, TKey> 方法调用。

请参阅

C# 参考

查询关键字（C# 参考）

LINQ 查询表达式（C# 编程指南）

group 子句（C# 参考）

Getting Started with LINQ in C#

join 子句（C# 参考）

使用 **join** 子句可以将来自不同源序列并且在对象模型中没有直接关系的元素相关联。唯一的要求是每个源中的元素需要共享某个可以进行比较以判断是否相等的值。例如，食品经销商可能具有某种产品的供应商列表以及买主列表。例如，可以使用 **join** 子句创建该产品同一指定地区供应商和买主的列表。

join 子句接受两个源序列作为输入。每个序列中的元素都必须是可以与另一个序列中的相应属性进行比较的属性，或者包含一个这样的属性。 **join** 子句使用特殊的 **equals** 关键字比较指定的键是否相等。 **join** 子句执行的所有联接都是同等联接。**join** 子句的输出形式取决于所执行的联接的具体类型。以下是三种最常见的联接类型：

- 内部联接
- 分组联接
- 左外部联接

内部联接

下面的示例演示一个简单的内部同等联接。此查询产生一个“产品名称/类别”对平面序列。同一类别字符串将出现在多个元素中。如果 categories 中的某个元素不具有匹配的 products，则该类别不会出现在结果中。

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name
```

有关更多信息，请参见 如何：执行内部联接（C# 编程指南）。

Group Join

含有 **into** 表达式的 **join** 子句称为分组联接。

```
var innerGroupJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID int
    select new { CategoryName = category.Name, Products = prodGrou
```

分组联接会产生一个分层的结果序列，该序列将左侧源序列中的元素与右侧源序列中的一个或多个匹配元素相关联。分组联接没有等效的关系术语；它本质上是一个对象数组序列。

如果在右侧源序列中找不到与左侧源中的元素相匹配的元素，则 **join** 子句会为该项产生一个空数组。因此，分组联接基本上仍然是一种内部同等联接，区别只在于分组联接将结果序列组织为多个组。

如果您只选择分组联接的结果，则可以访问各个项，但无法识别结果所匹配的键。因此，通常更为有用的做法是选择分组联接的结果并放入一个也具有该键名的新类型中，如上一个示例所示。

当然，还可以将分组联接的结果用作其他子查询的生成器：

```
var innerGroupJoinQuery2 =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID int
    from prod2 in prodGroup
    where prod2.UnitPrice > 2.50M
    select prod2;
```

有关更多信息，请参见 如何：执行分组联接（C# 编程指南）。

左外部联接

在左外部联接中，将返回左侧源序列中的所有元素，即使它们在右侧序列中没有匹配的元素也是如此。若要在 LINQ 中执行左外部联接，请将 **DefaultIfEmpty** 方法与分组联接结合起来，以指定要在某个左侧元素不具有匹配元素时产生的默认右侧元素。可以使用 **null** 作为任何引用类型的默认值，也可以指定用户定义的默认类型。下面的示例演示了用户定义的默认类型：

```
var leftOuterJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID int
    from item in prodGroup.DefaultIfEmpty(new Product { Name = Str
    select new { CatName = category.Name, ProdName = item.Name };
```

有关更多信息，请参见 如何：执行左外部联接（C# 编程指南）。

equals 运算符

join 子句执行同等联接。换句话说，只能基于两个键之间的相等关系进行匹配。其他类型的比较（例如，“greater than”或“not equals”）不受支持。为了表明所有联接都是同等联接，**join** 子句使用 **equals** 关键字而不是 **==** 运算符。 **equals** 关键字只

能用在 **join** 子句中，并且它与 **==** 运算符之间存在一个重要区别。对于 **equals**，左键使用外部源序列，而右键使用内部源序列。外部源仅在 **equals** 的左侧位于范围内，而内部源序列仅在其右侧位于范围内。

非同等联接

通过使用多个 **from** 子句将新序列单独引入到查询中，可以执行非同等联接、交叉联接和其他自定义联接操作。有关更多信息，请参见 如何：执行自定义联接操作（C# 编程指南）。

对象集合联接与关系表

在 LINQ 查询表达式中，联接操作是在对象集合上执行的。不能使用与两个关系表完全相同的方式“联接”对象集合。在 LINQ 中，仅当两个源序列没有通过任何关系相互联系时，才需要使用显式 **join** 子句。使用 LINQ to SQL 时，外键表在对象模型中表示为主表的属性。例如，在 Northwind 数据库中，Customer 表与 Orders 表之间具有外键关系。在将这两个表映射到对象模型时，Customer 类具有一个 Orders 属性，该属性包含与该 Customer 相关联的 Orders 的集合。实际上，已经为您执行了联接。

有关在 LINQ to SQL 的上下文中跨相关表执行查询的更多信息，请参见如何：映射数据库关系。

复合键

使用复合键可以测试多个值是否相等。有关更多信息，请参见 如何：使用复合键进行联接（C# 编程指南）。还可以在 **group** 子句中使用组合键。

下面的示例比较了使用相同的匹配键对相同数据源执行内部联接、分组联接和左外部联接的结果。这些示例中添加了一些额外的代码，以便在控制台显示中阐明结果。

```
class JoinDemonstration
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
```

```
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category(){Name="Beverages", ID=001},
        new Category(){ Name="Condiments", ID=002},
        new Category(){ Name="Vegetables", ID=003},
        new Category() {  Name="Grains", ID=004},
        new Category() {  Name="Fruit", ID=005}
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()
   {
      new Product{Name="Cola",  CategoryID=001},
      new Product{Name="Tea",  CategoryID=001},
      new Product{Name="Mustard", CategoryID=002},
      new Product{Name="Pickles", CategoryID=002},
      new Product{Name="Carrots", CategoryID=003},
      new Product{Name="Bok Choy", CategoryID=003},
      new Product{Name="Peaches", CategoryID=005},
      new Product{Name="Melons", CategoryID=005},
    };
    #endregion

    static void Main(string[] args)
    {
        JoinDemonstration app = new JoinDemonstration();

        app.InnerJoin();
        app.GroupJoin();
        app.GroupInnerJoin();
        app.GroupJoin3();
        app.LeftOuterJoin();
        app.LeftOuterJoin2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    void InnerJoin()
    {
        // Create the query that selects
        // a property from each element.
        var innerJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.Categor
           select new { Category = category.ID, Product = prod.Name

        Console.WriteLine("InnerJoin:");
        // Execute the query. Access results
```

```
        // with a simple foreach statement.
        foreach (var item in innerJoinQuery)
        {
            Console.WriteLine("{0,-10}{1}", item.Product, item.Cate
        }
        Console.WriteLine("InnerJoin: {0} items in 1 group.", inner
        Console.WriteLine(System.Environment.NewLine);

    }

    void GroupJoin()
    {
        // This is a demonstration query to show the output
        // of a "raw" group join. A more typical group join
        // is shown in the GroupInnerJoin method.
        var groupJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.Categor
           select prodGroup;

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Simple GroupJoin:");

        // A nested foreach statement is required to access group i
        foreach (var prodGrouping in groupJoinQuery)
        {
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("   {0,-10}{1}", item.Name, item.
            }
        }
        Console.WriteLine("Unshaped GroupJoin: {0} items in {1} unn
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupInnerJoin()
    {
        var groupJoinQuery2 =
            from category in categories
            orderby category.ID
            join prod in products on category.ID equals prod.Catego
            select new
            {
                Category = category.Name,
                Products = from prod2 in prodGroup
                           orderby prod2.Name
                           select prod2
            };
```

```
        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupInnerJoin:");
        foreach (var productGroup in groupJoinQuery2)
        {
            Console.WriteLine(productGroup.Category);
            foreach (var prodItem in productGroup.Products)
            {
                totalItems++;
                Console.WriteLine("  {0,-10} {1}", prodItem.Name, p
            }
        }
        Console.WriteLine("GroupInnerJoin: {0} items in {1} named g
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin3()
    {

        var groupJoinQuery3 =
            from category in categories
            join product in products on category.ID equals product.
            from prod in prodGroup
            orderby prod.CategoryID
            select new { Category = prod.CategoryID, ProductName = 

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupJoin3:");
        foreach (var item in groupJoinQuery3)
        {
            totalItems++;
            Console.WriteLine("   {0}:{1}", item.ProductName, item.
        }

        Console.WriteLine("GroupJoin3: {0} items in 1 group", total
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin()
    {
        // Create the query.
        var leftOuterQuery =
           from category in categories
           join prod in products on category.ID equals prod.Categor
           select prodGroup.DefaultIfEmpty(new Product() { Name = '

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Left Outer Join:");
```

```
        // A nested foreach statement  is required to access group
        foreach (var prodGrouping in leftOuterQuery)
        {
            Console.WriteLine("Group:", prodGrouping.Count());
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("  {0,-10}{1}", item.Name, item.C
            }
        }
        Console.WriteLine("LeftOuterJoin: {0} items in {1} groups",
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin2()
    {
        // Create the query.
        var leftOuterQuery2 =
           from category in categories
           join prod in products on category.ID equals prod.Categor
           from item in prodGroup.DefaultIfEmpty()
           select new { Name = item == null ? "Nothing!" : item.Nam

        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", l
        // Store the count of total items
        int totalItems = 0;

        Console.WriteLine("Left Outer Join 2:");

        // Groups have been flattened.
        foreach (var item in leftOuterQuery2)
        {
            totalItems++;
            Console.WriteLine("{0,-10}{1}", item.Name, item.Categor
        }
        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", t
    }
}
/*Output:

InnerJoin:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Peaches   5
Melons    5
InnerJoin: 8 items in 1 group.

Unshaped GroupJoin:
```

```
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
Group:
    Peaches   5
    Melons    5
Unshaped GroupJoin: 8 items in 5 unnamed groups

GroupInnerJoin:
Beverages
    Cola       1
    Tea        1
Condiments
    Mustard    2
    Pickles    2
Vegetables
    Bok Choy   3
    Carrots    3
Grains
Fruit
    Melons     5
    Peaches    5
GroupInnerJoin: 8 items in 5 named groups

GroupJoin3:
    Cola:1
    Tea:1
    Mustard:2
    Pickles:2
    Carrots:3
    Bok Choy:3
    Peaches:5
    Melons:5
GroupJoin3: 8 items in 1 group

Left Outer Join:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
```

```
    Nothing!  4
Group:
    Peaches   5
    Melons    5
LeftOuterJoin: 9 items in 5 groups

LeftOuterJoin2: 9 items in 1 group
Left Outer Join 2:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Nothing!  4
Peaches   5
Melons    5
LeftOuterJoin2: 9 items in 1 group
Press any key to exit.
*/
```

备注

后面未跟 **into** 的 **join** 子句被转换为 Join<TOuter, TInner, TKey, TResult> 方法调用；后面跟有 **into** 的 **join** 子句被转换为 GroupJoin<TOuter, TInner, TKey, TResult> 方法调用。

请参阅

查询关键字（C# 参考）

LINQ 查询表达式（C# 编程指南）

Join Operations

group 子句（C# 参考）

如何：执行左外部联接（C# 编程指南）

如何：执行内部联接（C# 编程指南）

如何：执行分组联接（C# 编程指南）

如何：对 Join 子句的结果进行排序（C# 编程指南）

如何：使用复合键进行联接（C# 编程指南）

如何：安装示例数据库

let 子句（C# 参考）

在查询表达式中，存储子表达式的结果有时很有用，这样可以在随后的子句中使用。可以使用 **let** 关键字完成这一工作，该关键字可以创建一个新的范围变量，并且用您提供的表达式的结果初始化该变量。一旦用值初始化了该范围变量，它就不能用于存储其他值。但如果该范围变量存储的是可查询的类型，则可以对其进行查询。

在下面的示例中，以两种方式使用了 **let**：

1. 创建一个可以查询自身的可枚举类型。
2. 使查询只能对范围变量 word 调用一次 ToLower。如果不使用 **let**，则必须在 **where** 子句的每个谓词中调用 ToLower。

```
class LetSample1
{
    static void Main()
    {
        string[] strings =
        {
            "A penny saved is a penny earned.",
            "The early bird catches the worm.",
            "The pen is mightier than the sword."
        };

        // Split the sentence into an array of words
        // and select those whose first letter is a vowel.
        var earlyBirdQuery =
            from sentence in strings
            let words = sentence.Split(' ')
            from word in words
            let w = word.ToLower()
            where w[0] == 'a' || w[0] == 'e'
                || w[0] == 'i' || w[0] == 'o'
                || w[0] == 'u'
            select word;

        // Execute the query.
        foreach (var v in earlyBirdQuery)
        {
            Console.WriteLine("\"{0}\" starts with a vowel", v);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    "A" starts with a vowel
    "is" starts with a vowel
    "a" starts with a vowel
    "earned." starts with a vowel
    "early" starts with a vowel
    "is" starts with a vowel
*/
```

请参阅

C# 参考

查询关键字（C# 参考）

LINQ 查询表达式（C# 编程指南）

Getting Started with LINQ in C#

如何：在查询表达式中处理异常（C# 编程指南）

ascending（C# 参考）

ascending 上下文关键字用在查询表达式的 orderby 子句中，用于指定从最小到最大的排序顺序。因为 **ascending** 是默认排序顺序，所以您无须指定它。

下面的示例演示 **ascending** 在 orderby 子句中的用法。

```
IEnumerable<string> sortAscendingQuery =
    from vegetable in vegetables
    orderby vegetable ascending
    select vegetable;
```

请参阅

C# 参考

LINQ 查询表达式（C# 编程指南）

descending（C# 参考）

descending（C# 参考）

descending 上下文关键字用在查询表达式的 orderby 子句中，用于指定从最大到最小的排序顺序。

下面的示例演示 **descending** 在 orderby 子句中的用法。

```
IEnumerable<string> sortDescendingQuery =
    from vegetable in vegetables
    orderby vegetable descending
    select vegetable;
```

请参阅

C# 参考

LINQ 查询表达式（C# 编程指南）

ascending（C# 参考）

on（C# 参考）

on 上下文关键字在查询表达式的 join 子句中用于指定联接条件。

下面的示例演示 **on** 在 **join** 子句中的用法。

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name
```

请参阅

C# 参考

LINQ 查询表达式（C# 编程指南）

equals（C# 参考）

equals 上下文关键字用在查询表达式的 **join** 子句中，用于比较两个序列的元素。有关更多信息，请参见 join 子句（C# 参考）。

下面的示例演示 **equals** 关键字在 **join** 子句中的用法。

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name
```

请参阅

LINQ 查询表达式（C# 编程指南）

by（C# 参考）

by 上下文关键字用在查询表达式的 **group** 子句中，用于指定应该如何对返回的项进行分组。有关更多信息，请参见 group 子句（C# 参考）。

下面的示例演示如何在 **group** 子句中使用 **by** 上下文关键字，以指定按照每个学生姓氏的第一个字母对学生进行分组。

```
var query = from student in students
            group student by student.LastName[0];
```

请参阅

LINQ 查询表达式（C# 编程指南）

in（C# 参考）

in 上下文关键字可在下面三种上下文中使用：

- foreach 语句
- 查询表达式中的 join 子句
- 泛型接口和委托中的泛型类型参数。

请参阅

C# 参考

C# 运算符

C# 提供了许多运算符，这些运算符是指定要在表达式中执行哪些操作（数学、索引、函数调用等等）的符号。在应用于用户定义类型之前，你可以对许多运算符进行重载以更改其含义。

针对整型类型的运算（如 **==**、**!=**、**<**、**>**、**&**、**|**）通常可用于枚举 (**enum**) 类型。

以下章节按最高优先级到最低优先级的顺序列示 C# 运算符。各章节内运算符的优先级相同。

主要运算符

以下是具有最高优先级的运算符。请注意，你可以单击运算符转到包含示例的详细页面。

x.y – 成员访问。

x?.y – null 条件成员访问。如果左边操作数为 null，则返回 null。

f(x) – 函数调用。

a[x] – 聚合对象索引。

a?[x] – null 条件索引。如果左边操作数为 null，则返回 null。

x++ – 后缀递增。先返回 x 值，然后用加 1（通常加整数 1）后的 x 值更新存储位置。

x-- – 后缀递减。先返回 x 值，然后用减 1（通常减整数 1）后的 x 值更新存储位置。

New – 类型实例化。

Typeof – 返回表示操作数的 System.Type 对象。

Checked – 对整数运算启用溢出检查。

Unchecked – 对整数运算禁用溢出检查。这是默认的编译器行为。

default(T) – 返回类型 T 的默认初始化值，T 为引用类型时返回 null，T 为数值类型时返回零，T 为结构类型时返回填充为零/null 的成员。

Delegate – 声明并返回一个委托实例。

Sizeof – 返回类型操作数的大小（以字节为单位）。

-> – 取消指针引用与成员访问相结合。

一元运算符

这些运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击运算符转到包含示例的详细页面。

+x – 返回 x 值。

-x – 数值求反。

!x – 逻辑求反。

~x – 按位求补。

++x – 前缀递增。先用加 1（通常加整数 1）后的 x 值更新存储位置，然后返回 x 值。

--x – 前缀递减。先用减 1（通常减整数 1）后的 x 值更新存储位置，然后返回 x 值。

(T)x – 类型转换。

Await – 等待 Task。

&x – 地址。

*x – 取消引用。

乘法运算符

这些运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击运算符转到包含示例的详细页面。

x * y – 乘法。

x / y – 除法。如果操作数均为整数，则结果为整数，舍去小数（例如，-7 / 2 is -3）。

x % y – 取模。如果操作数均为整数，则返回 x 除以 y 后的余数。如果 q = x / y 且 r = x % y，则 x = q * y + r。

加法运算符

这些运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击运算符转到包含示例的详细页面。

x + y – 加法。

x – y – 减法。

移位运算符

这些运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击运算符转到包含示例的详细页面。

x << y – 向左移位，右边移出的空位补零。

x >> y – 向右移位。如果左操作数是 **int** 或 **long**，则左位数补符号位。如果左操作数是 **uint** 或 **ulong**，则左位数补零。

关系和类型测试运算符

这些运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击运算符转到包含示例的详细页面。

x < y – 小于（如果 x 小于 y，则为 true）。

x > y – 大于（如果 x 大于 y，则为 true）。

x <= y – 小于等于。

x >= y – 大于等于。

Is – 类型兼容性。如果求值后的左操作数可以转换为右操作数中指定的类型（静态类型），则返回 true。

As – 类型转换。返回左操作数并转换为右操作数中指定的类型（静态类型），但 **as** 返回 **null**，其中 (T)x 会引发异常。

相等运算符

这些运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击运算符转到包含示例的详细页面。

x == y – 相等。默认情况下，对于 **string** 以外的引用类型，此运算符返回引用相等（标识测试）。但是，类型可以重载 **==**，因此，如果你想测试标识，最好对 **object** 使用 ReferenceEquals 方法。

x != y – 不相等。请参阅有关 **==** 的注释。如果某个类型重载 **==**，则它必须重载 **!=**。

逻辑 AND 运算符

此运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击该运算符转到包含示例的详细页面。

x & y – 逻辑或位 AND。与整数类型一起使用，并且通常允许 **enum** 类型。

逻辑 XOR 运算符

此运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击该运算符转到包含示例的详细页面。

x ^ y – 逻辑或位 XOR。通常可以将此运算符与整数类型和 **enum** 类型一起使用。

逻辑 OR 运算符

此运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击该运算符转到包含示例的详细页面。

x | y – 逻辑或位 OR。与整数类型一起使用，并且通常允许 **enum** 类型。

条件 AND 运算符

此运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击该运算符转到包含示例的详细页面。

x && y – 逻辑 AND。如果第一个操作数为 false，则 C# 不对第二个操作数求值。

条件 OR 运算符

此运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击该运算符转到包含示例的详细页面。

x || y – 逻辑 OR。如果第一个操作数为 true，则 C# 不对第二个操作数求值。

Null 合并运算符

此运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击该运算符转到包含示例的详细页面。

x ?? y – 如果不为 **null**，则返回 x；否则返回 y。

条件运算符

此运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击该运算符转到包含示例的详细页面。

t ? x : y – 如果测试 t 为 true，则求值并返回 x；否则，求值并返回 y。

赋值和 Lambda 运算符

这些运算符的优先级比下一章节高，比上一章节低。请注意，你可以单击运算符转到包含示例的详细页面。

x = y – 赋值。

x += y – 递增。 x 值加 y 值，结果存储在 x 中，并返回新值。如果 x 指定 **event**，则 y 必须是 C# 作为事件处理程序添加的相应函数。

x -= y – 递减。 x 值减 y 值，结果存储在 x 中，并返回新值。如果 x 指定 **event**，则 y 必须是 C# 作为事件处理程序删除的相应函数

x *= y – 乘法赋值。 x 值乘以 y 值，结果存储在 x 中，并返回新值。

x /= y – 除法赋值。 x 值除以 y 值，结果存储在 x 中，并返回新值。

x %= y – 取模赋值。 x 值除以 y 值，余数存储在 x 中，并返回新值。

x &= y – AND 赋值。 y 值和 x 值相与，结果存储在 x 中，并返回新值。

x |= y – OR 赋值。 y 值和 x 值相或，结果存储在 x 中，并返回新值。

x ^= y – XOR 赋值。 y 值和 x 值相异或，结果存储在 x 中，并返回新值。

x <<= y – 左移赋值。将 x 值向左移动 y 位，结果存储在 x 中，并返回新值。

x >>= y – 右移赋值。将 x 值向右移动 y 位，结果存储在 x 中，并返回新值。

=> – lambda 声明。

算术溢出

算术运算符（+、-、*、/）产生的结果可能会超出所涉数值类型的可能值的范围。详细信息应参考特定运算符的相关章节，而一般情况下：

- 整数算术溢出或者引发 OverflowException，或者放弃结果的最高有效位。整数被零除总是引发 **DivideByZeroException**。
- 浮点算术溢出或被零除从不引发异常，因为浮点类型基于 IEEE 754，因此可以表示无穷大和 NaN（非数值）。
- 小数算术溢出总是引发 OverflowException。小数被零除总是引发 DivideByZeroException。

当发生整数溢出时，产生的结果取决于执行上下文，该上下文可为 checked 或 unchecked。在 checked 上下文中引发 OverflowException。在 unchecked 上下文中，放弃结果的最高有效位并继续执行。因此，C# 让你有机会选择处理或忽略溢出。

除算术运算符以外，整型类型之间的转换也会导致溢出（例如，将 long 转换为 int）并受 checked 或 unchecked 执行的限制。但是，位运算符和移位运算符永远不会导致溢出。

请参阅

C# 参考

C# 编程指南

Visual C#

可重载运算符（C# 编程指南）

C# 关键字

运算符（C# 参考）

方括号 ([]) 用于数组、索引器和特性，也可用于指针。

备注

数组类型是一种后跟 [] 的类型：

```
int[] fib; // fib is of type int[], "array of int".
fib = new int[100]; // Create a 100-element int array.
```

若要访问数组的一个元素，则用方括号括起所需元素的索引：

```
fib[0] = fib[1] = 1;
for (int i = 2; i < 100; ++i) fib[i] = fib[i - 1] + fib[i - 2];
```

如果数组索引超出范围，则会引发异常。

不能重载数组索引运算符；但类型可以定义采用一个或多个参数的索引器和属性。索引器参数括在方括号中，这一点与数组索引类似；但可以将索引器参数声明为任何类型，这一点与数组索引不同，后者必须为整形。

例如，.NET Framework 定义 **Hashtable** 类型，该类型将键和任意类型的值关联在一起。

```
System.Collections.Hashtable h = new System.Collections.Hashtable()
h["a"] = 123; // Note: using a string as the index.
```

方括号还用于指定 特性（C# 和 Visual Basic）：

```
// using System.Diagnostics;
[Conditional("DEBUG")]
void TraceMethod() {}
```

可以使用方括号来指定指针索引：

```
unsafe void M()
{
    int[] nums = {0,1,2,3,4,5};
    fixed ( int* p = nums )
    {
        p[0] = p[1] = 1;
        for( int i=2; i<100; ++i ) p[i] = p[i-1] + p[i-2];
    }
}
```

不执行边界检查。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 运算符

数组（C# 编程指南）

索引器（C# 编程指南）

unsafe（C# 参考）

fixed 语句（C# 参考）

() 运算符（C# 参考）

除了用于指定表达式中运算符的顺序外，圆括号还用于执行以下任务：

1. 指定强制转换或类型转换。

```
double x = 1234.7;
int a;
a = (int)x; // Cast double to int
```

2. 调用方法或委托。

```
TestMethod();
```

备注

强制转换显式调用从一种类型到另一种类型的转换运算符；如果未定义这样的转换运算符，则强制转换将失败。若要定义转换运算符，请参见 explicit 和 implicit。

不能重载 () 运算符。

有关更多信息，请参见 强制转换和类型转换（C# 编程指南）。

强制转换表达式可能会使语法发生歧义。

例如，表达式(x)–y既可以解释为强制转换表达式 （ 将–y强制转换为类型x ）， 也可以解释为带括号的相加表达式（计算x–y的值）。

有关方法调用的详细信息，请参阅方法（C# 编程指南）。

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 运算符

. 运算符（C# 参考）

点运算符 (.) 用于成员访问。点运算符指定类型或命名空间的成员。例如，点运算符用于访问 .NET Framework 类库中的特定方法：

```
// The class Console in namespace System:
System.Console.WriteLine("hello");
```

例如，请考虑以下类：

```
  class Simple
{
    public int a;
    public void b()
    {
    }
}
```

```
Simple s = new Simple();
```

变量 s 有两个成员 a 和 b；若要访问这两个成员，请使用点运算符：

```
s.a = 6;   // assign to field a;
s.b();     // invoke member function b;
```

点还用于构造限定名，即指定其所属的命名空间或接口的名称。

```
// The class Console in namespace System:
System.Console.WriteLine("hello");
```

using 指令使某些名称限定可选：

```
namespace ExampleNS
{
    using System;
    class C
    {
        void M()
        {
            System.Console.WriteLine("hello");
            Console.WriteLine("hello");   // Same as previous line
        }
    }
}
```

但是当某一标识符不确定时，必须限定它：

```
namespace Example2
{
    class Console
    {
        public static void WriteLine(string s){}
    }
}
namespace Example1
{
    using System;
    using Example2;
    class C
    {
        void M()
        {
            // Console.WriteLine("hello");   // Compiler error. Amb
            System.Console.WriteLine("hello"); //OK
            Example2.Console.WriteLine("hello"); //OK
        }
    }
}
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 运算符

:: 运算符（C# 参考）

命名空间别名限定符 (::) 用于查找标识符。它通常放置在两个标识符之间，例如：

```
global::System.Console.WriteLine("Hello World");
```

备注

命名空间别名限定符可以是 global。这将调用全局命名空间中的查找，而不是在别名命名空间中。

更多信息

有关如何使用 :: 运算符的示例，请参见下面的章节：

- 如何：使用全局命名空间别名（C# 编程指南）

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 运算符

命名空间关键字（C# 参考）

. 运算符（C# 参考）

外部别名（C# 参考）

+ 运算符（C# 参考）

+ 运算符既可作为一元运算符也可作为二元运算符。

备注

一元 + 运算符是为所有数值类型预定义的。对数值类型进行一元 + 运算的结果就是操作数的值。

为数值类型和字符串类型预定义了二元 + 运算符。对于数值类型，+ 计算两个操作数之和。当其中的一个操作数是字符串类型或两个操作数都是字符串类型时，+ 将操作数的字符串表示形式串联在一起。

委托类型也提供二元 + 运算符，该运算符执行委托串联。

用户定义的类型可重载一元 + 运算符和二元 + 运算符。 对于整数类型适用的运算对枚举类型通常也适用。 有关更多信息，请参见 运算符（C# 参考）。

```
class Plus
{
    static void Main()
    {
        Console.WriteLine(+5);        // unary plus
        Console.WriteLine(5 + 5);     // addition
        Console.WriteLine(5 + .5);    // addition
        Console.WriteLine("5" + "5"); // string concatenation
        Console.WriteLine(5.0 + "5"); // string concatenation
        // note automatic conversion from double to string
    }
}
/*
Output:
5
10
5.5
55
55
*/
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 运算符

运算符（C# 参考）

- 运算符（C# 参考）

- 运算符既可作为一元运算符也可作为二元运算符。

备注

一元 - 运算符是为所有数值类型预定义的。数值类型的一元 - 运算的结果是操作数的反数。

二元 - 运算符是为所有数值类型和枚举类型预定义的，其功能是从第一个操作数中减去第二个操作数。

委托类型也提供二元 - 运算符，该运算符执行委托移除。

用户定义的类型可重载一元 - 运算符和二元 - 运算符。有关更多信息，请参见 运算符（C# 参考）。

```
class MinusLinus
{
    static void Main()
    {
        int a = 5;
        Console.WriteLine(-a);
        Console.WriteLine(a - 1);
        Console.WriteLine(a - .5);
    }
}
/*
Output:
-5
4
4.5
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

* 运算符（C# 参考）

乘法运算符 (*)，用于计算操作数的积。另外还用作取消引用运算符，允许读取和写入指针。

备注

所有数值类型都具有预定义的乘法运算符。

* 运算符还用来声明指针类型和取消引用指针。该运算符只能在不安全的上下文中使用，通过 unsafe 关键字的使用来表示，并且需要 /unsafe 编译器选项。取消引用运算符也称为间接寻址运算符。

用户定义的类型可重载二元 * 运算符（请参见 operator）。重载二元运算符时，也会隐式重载相应的赋值运算符（如果有）。

```
class Multiply
{
    static void Main()
    {
        Console.WriteLine(5 * 2);
        Console.WriteLine(-.5 * .2);
        Console.WriteLine(-.5m * .2m); // decimal type
    }
}
    /*
Output
    10
    -0.1
    -0.10
*/
```

```
public class Pointer
{
    unsafe static void Main()
    {
        int i = 5;
        int* j = &i;
        System.Console.WriteLine(*j);
    }
}
/*
Output:
5
*/
```

请参阅

C# 参考

C# 编程指南

不安全代码和指针（C# 编程指南）

C# 运算符

/ 运算符（C# 参考）

除法运算符 (/) 将通过第二个操作数的第一个操作数。 所有数值类型都具有预定义的除法运算符。

备注

用户定义的类型可重载 **/** 运算符（请参见运算符）。 对 **/** 运算符进行重载将隐式重载 /= 运算符。

两个整数相除的结果始终为一个整数。 例如，7 的结果 / 3 是 2。 要确定 7 的其余部分 / 3，使用余数运算符 (%）。 若要获取作为有理数或分数的商，应将被除数或除数设置为 **float** 类型或 **double** 类型。 如果您通过将数字放到小数点的右侧，如以下示例所示表示被除数或除数为小数，您可以隐式指定类型。

示例

```
class Division
{
    static void Main()
    {
        Console.WriteLine("\nDividing 7 by 3.");
        // Integer quotient is 2, remainder is 1.
        Console.WriteLine("Integer quotient:           {0}", 7 / 3
        Console.WriteLine("Negative integer quotient:  {0}", -7 / 
        Console.WriteLine("Remainder:                  {0}", 7 % 3
        // Force a floating point quotient.
        float dividend = 7;
        Console.WriteLine("Floating point quotient:    {0}", divide

        Console.WriteLine("\nDividing 8 by 5.");
        // Integer quotient is 1, remainder is 3.
        Console.WriteLine("Integer quotient:           {0}", 8 / 5
        Console.WriteLine("Negative integer quotient:  {0}", 8 / -
        Console.WriteLine("Remainder:                  {0}", 8 % 5
        // Force a floating point quotient.
        Console.WriteLine("Floating point quotient:    {0}", 8 / 5
    }
}
// Output:
//Dividing 7 by 3.
//Integer quotient:           2
//Negative integer quotient:  -2
//Remainder:                  1
//Floating point quotient:    2.33333333333333

//Dividing 8 by 5.
//Integer quotient:           1
//Negative integer quotient:  -1
//Remainder:                  3
//Floating point quotient:    1.6
```

请参阅

C# 参考

C# 编程指南

C# 运算符

% 运算符（C# 参考）

% 运算符在将其第一个操作数后计算其余部分在其秒钟之前。所有数字类型预定义的其余部分运算符。

备注

用户定义的类型能重载 **%** 运算符 (请参见 运算符)。当重载时二元运算符时，，如果有，还将隐式重载对应的赋值运算符。

```
class MainClass6
{
    static void Main()
    {
        Console.WriteLine(5 % 2);       // int
        Console.WriteLine(-5 % 2);      // int
        Console.WriteLine(5.0 % 2.2);   // double
        Console.WriteLine(5.0m % 2.2m); // decimal
        Console.WriteLine(-5.2 % 2.0);  // double
    }
}
/*
Output:
1
-1
0.6
0.6
-1.2
*/
```

注释

请注意舍入误差与双精度类型。

请参阅

C# 参考

C# 编程指南

C# 运算符

& 运算符（C# 参考）

& 运算符既可作为一元运算符也可作为二元运算符。

备注

一元 & 运算符返回操作数的地址（要求 unsafe 上下文）。

为整型和 **bool** 类型预定义了二进制 & 运算符。对于整型，& 计算操作数的逻辑按位“与”。对于 **bool** 操作数，& 计算操作数的逻辑“与”；也就是说，当且仅当两个操作数均为 **true** 时，结果才为 **true**。

& 运算符计算两个运算符，与第一个操作数的值无关。例如：

```
int i = 0;
if (false & ++i == 1)
{
    // i is incremented, but the conditional
    // expression evaluates to false, so
    // this block does not execute.
}
```

用户定义的类型可重载二元 **&** 运算符（请参见 operator）。 对于整数类型适用的运算对枚举类型通常也适用。 重载二元运算符时，也会隐式重载相应的赋值运算符（如果有）。

```
class BitwiseAnd
{
    static void Main()
    {
        // The following two statements perform logical ANDs.
        Console.WriteLine(true & false);
        Console.WriteLine(true & true);

        // The following line performs a bitwise AND of F8 (1111 10
        // 3F (0011 1111).
        //    1111 1000
        //    0011 1111
        //    ---------
        //    0011 1000 or 38
        Console.WriteLine("0x{0:x}", 0xf8 & 0x3f);
    }
}
// Output:
// False
// True
// 0x38
```

请参阅

C# 参考

C# 编程指南

C# 运算符

| 运算符（C# 参考）

Binary| 运算符是为整型和 **bool** 预定义的。对于整型，**|**计算操作数的按位“或”。对于 **bool** 操作数，**|** 计算操作数的逻辑“或”；也就是说，当且仅当两个操作数均为 **false** 时，结果才为 **false**。

备注

用户定义的类型可重载**|** 运算符（请参见运算符）。

```
class OR
{
    static void Main()
    {
        Console.WriteLine(true | false);  // logical or
        Console.WriteLine(false | false); // logical or
        Console.WriteLine("0x{0:x}", 0xf8 | 0x3f);   // bitwise or
    }
}
/*
Output:
True
False
0xff
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

^ 运算符（C# 参考）

二元 ^ 运算符是为整型和 **bool** 类型预定义的。对于整型，^ 将计算操作数的按位“异或”。对于 **bool** 操作数，^ 将计算操作数的逻辑“异或”；也就是说，当且仅当只有一个操作数为 **true** 时，结果才为 **true**。

备注

用户定义的类型可重载 ^ 运算符（请参见运算符）。 对于整数类型适用的运算对枚举类型通常也适用。

```
class XOR
{
    static void Main()
    {
        // Logical exclusive-OR

        // When one operand is true and the other is false, exclusi
        // returns True.
        Console.WriteLine(true ^ false);
        // When both operands are false, exclusive-OR returns False
        Console.WriteLine(false ^ false);
        // When both operands are true, exclusive-OR returns False.
        Console.WriteLine(true ^ true);

        // Bitwise exclusive-OR

        // Bitwise exclusive-OR of 0 and 1 returns 1.
        Console.WriteLine("Bitwise result: {0}", Convert.ToString((
        // Bitwise exclusive-OR of 0 and 0 returns 0.
        Console.WriteLine("Bitwise result: {0}", Convert.ToString((
        // Bitwise exclusive-OR of 1 and 1 returns 0.
        Console.WriteLine("Bitwise result: {0}", Convert.ToString((

        // With more than one digit, perform the exclusive-OR colum
        //    10
        //    11
        //    --
        //    01
        // Bitwise exclusive-OR of 10 (2) and 11 (3) returns 01 (1)
        Console.WriteLine("Bitwise result: {0}", Convert.ToString((

        // Bitwise exclusive-OR of 101 (5) and 011 (3) returns 110
        Console.WriteLine("Bitwise result: {0}", Convert.ToString((

        // Bitwise exclusive-OR of 1111 (decimal 15, hexadecimal F)
        // returns 1010 (decimal 10, hexadecimal A).
```

```
        Console.WriteLine("Bitwise result: {0}", Convert.ToString((

        // Finally, bitwise exclusive-OR of 11111000 (decimal 248,
        // and 00111111 (decimal 63, hexadecimal 3F) returns 110001
        // 199 in decimal, C7 in hexadecimal.
        Console.WriteLine("Bitwise result: {0}", Convert.ToString((
    }
}
/*
Output:
True
False
False
Bitwise result: 1
Bitwise result: 0
Bitwise result: 0
Bitwise result: 1
Bitwise result: 110
Bitwise result: 1010
Bitwise result: 11000111
*/
```

在前面的示例中，0xf8 ^ 0x3f 的计算对以下两个二进制值（分别对应于十六进制值 F8 和 3F）执行按位“异或”运算：

1111 1000

0011 1111

“异或”运算的结果是 1100 0111，即十六进制值 C7。

请参阅

C# 参考

C# 编程指南

C# 运算符

! 运算符（C# 参考）

逻辑非运算符 (!) 是对操作数求反的一元运算符。为 **bool** 定义了该运算符，当且仅当操作数为 **false** 时才返回 **true**。

备注

用户定义的类型可重载 **!** 运算符（请参见运算符）。

```
class MainClass4
{
    static void Main()
    {
        Console.WriteLine(!true);
        Console.WriteLine(!false);
    }
}
/*
Output:
False
True
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

~ 运算符（C# 参考）

~ 运算符对操作数执行按位求补运算，其效果相当于反转每一位。按位求补运算符是为 int、uint、long 和 ulong 类型预定义的。

注意
~ 符合也用来声明析构函数。有关更多信息，请参见 析构函数（C# 编程指南）。

备注

用户定义的类型可重载 ~ 运算符。有关更多信息，请参见 operator。 对于整数类型适用的运算对枚举类型通常也适用。

```
class BWC
{
    static void Main()
    {
        int[] values = { 0, 0x111, 0xfffff, 0x8888, 0x22000022 };
        foreach (int v in values)
        {
            Console.WriteLine("~0x{0:x8} = 0x{1:x8}", v, ~v);
        }
    }
}
/*
Output:
~0x00000000 = 0xffffffff
~0x00000111 = 0xfffffeee
~0x000fffff = 0xfff00000
~0x00008888 = 0xffff7777
~0x22000022 = 0xddffffdd
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

析构函数（C# 编程指南）

= 运算符（C# 参考）

赋值运算符 (=) 将右操作数的值存储在左操作数表示的存储位置、属性或索引器中，并将值作为结果返回。操作数的类型必须相同（即右操作数必须可以隐式转换为左操作数的类型）。

备注

不能重载赋值运算符。不过，可为类型定义隐式转换运算符，这样就可以对这些类型使用赋值运算符。有关更多信息，请参见 使用转换运算符（C# 编程指南）。

```
class Assignment
{
    static void Main()
    {
        double x;
        int i;
        i = 5; // int to int assignment
        x = i; // implicit conversion from int to double
        i = (int)x; // needs cast
        Console.WriteLine("i is {0}, x is {1}", i, x);
        object obj = i;
        Console.WriteLine("boxed value = {0}, type is {1}",
                  obj, obj.GetType());
        i = (int)obj;
        Console.WriteLine("unboxed: {0}", i);
    }
}
/*
Output:
i is 5, x is 5
boxed value = 5, type is System.Int32
unboxed: 5
 */
```

请参阅

C# 参考

C# 编程指南

C# 运算符

< 运算符（C# 参考）

所有数值和枚举类型都定义“小于”关系运算符 (**<**)，如果第一个操作数小于第二个操作数，该运算符返回 **true**，否则返回 **false**。

备注

用户定义的类型可重载 **<** 运算符（请参见运算符）。如果重载 **<**，则还必须重载 >。重载二元运算符时，也会隐式重载相应的赋值运算符（如果有）。

```
class LT
{
    static void Main()
    {
        Console.WriteLine(1 < 1.1);
        Console.WriteLine(1.1 < 1.1);
    }
}
/*
Output:
True
False
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

> 运算符（C# 参考）

所有数值类型和枚举类型都定义“大于”关系运算符 **>**，如果第一个操作数大于第二个操作数，它将返回 **true**，否则返回 **false**。

备注

用户定义的类型可重载 **>** 运算符（请参见运算符）。如果重载 **>**，则还必须重载 <。重载二元运算符时，也会隐式重载相应的赋值运算符（如果有）。

```
class GT
{
    static void Main()
    {
        Console.WriteLine(1.1 > 1);
        Console.WriteLine(1.1 > 1.1);
    }
}
/*
Output:
True
False
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

explicit（C# 参考）

?: 运算符（C# 参考）

条件运算符 (**?:**) 根据 Boolean 表达式的值返回两个值之一。下面是条件运算符的语法。

```
condition ? first_expression : second_expression;
```

备注

condition 的计算结果必须为 **true** 或 **false**。如果 *condition* 为 **true**，则将计算 *first_expression* 并使其成为结果。如果 *condition* 为 **false**，则将计算 *second_expression* 并使其成为结果。只计算两个表达式之一。

first_expression 和 *second_expression* 的类型必须相同，或者必须存在从一种类型到另一种类型的隐式转换。

你可通过使用条件运算符表达可能更确切地要求 **if-else** 构造的计算。例如，以下代码首先使用 **if** 语句，然后使用条件运算符将整数分类为正整数或负整数。

```
int input = Convert.ToInt32(Console.ReadLine());
string classify;

// if-else construction.
if (input > 0)
    classify = "positive";
else
    classify = "negative";

// ?: conditional operator.
classify = (input > 0) ? "positive" : "negative";
```

条件运算符为右联运算符。表达式 a ? b : c ? d : e 作为 a ? b : (c ? d : e) 而非 (a ? b : c) ? d : e 进行计算。

无法重载条件运算符。

```
class ConditionalOp
{
    static double sinc(double x)
    {
        return x != 0.0 ? Math.Sin(x) / x : 1.0;
    }

    static void Main()
    {
        Console.WriteLine(sinc(0.2));
        Console.WriteLine(sinc(0.1));
        Console.WriteLine(sinc(0.0));
    }
}
/*
Output:
0.993346653975306
0.998334166468282
1
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

if-else（C# 参考）

++ 运算符（C# 参考）

递增运算符 (**++**) 将操作数加 1。递增运算符可出现在操作数之前或之后：

备注

第一种形式是前缀增量操作。该操作的结果是操作数加 1 之后的值。

第二种形式是后缀增量操作。该运算的结果是操作数增加之前的值。

数值类型和枚举类型具有预定义的增量运算符。用户定义的类型可重载 **++** 运算符。 对于整数类型适用的运算对枚举类型通常也适用。

```
//++ operator
class MainClass
{
    static void Main()
    {
        double x;
        x = 1.5;
        Console.WriteLine(++x);
        x = 1.5;
        Console.WriteLine(x++);
        Console.WriteLine(x);
    }
}
/*
Output
2.5
1.5
2.5
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

-- 运算符（C# 参考）

递减运算符 (--) 将操作数减 1。递减运算符可出现在操作数之前或之后：--variable 和 variable--。第一种形式是前缀减量操作。该运算的结果是操作数减小“之后”的值。第二种形式是后缀减量操作。该运算的结果是操作数减小“之前”的值。

备注

数值类型和枚举类型具有预定义的增量运算符。

用户定义的类型可重载 -- 运算符（请参见运算符）。 对于整数类型适用的运算对枚举类型通常也适用。

```
class MainClass5
{
    static void Main()
    {
        double x;
        x = 1.5;
        Console.WriteLine(--x);
        x = 1.5;
        Console.WriteLine(x--);
        Console.WriteLine(x);
    }
}
/*
Output:
0.5
1.5
0.5
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

&& 运算符（C# 参考）

条件“与”运算符 (&&) 执行其 **bool** 操作数的逻辑“与”运算，但仅在必要时才计算第二个操作数。

备注

操作

```
x && y
```

对应于操作

```
x & y
```

，但，如果 x是 **false**， y 不会计算，因为，和操作的结果是 false ，无论 y 的值为。这被称作为“短路”计算。

不能重载条件“与”运算符，但常规逻辑运算符和运算符 true 与 false 的重载，在某些限制条件下也被视为条件逻辑运算符的重载。

在下面的示例中，，因为该操作数返回 false，在第二个 **if** 语句的条件表达式计算只有第一个操作数。

```
class LogicalAnd
{
    static void Main()
    {
        // Each method displays a message and returns a Boolean val
        // Method1 returns false and Method2 returns true. When & i
        // both methods are called.
        Console.WriteLine("Regular AND:");
        if (Method1() & Method2())
            Console.WriteLine("Both methods returned true.");
        else
            Console.WriteLine("At least one of the methods returned

        // When && is used, after Method1 returns false, Method2 is
        // not called.
        Console.WriteLine("\nShort-circuit AND:");
        if (Method1() && Method2())
            Console.WriteLine("Both methods returned true.");
        else
            Console.WriteLine("At least one of the methods returned
    }

    static bool Method1()
    {
        Console.WriteLine("Method1 called.");
        return false;
    }

    static bool Method2()
    {
        Console.WriteLine("Method2 called.");
        return true;
    }
}
// Output:
// Regular AND:
// Method1 called.
// Method2 called.
// At least one of the methods returned false.

// Short-circuit AND:
// Method1 called.
// At least one of the methods returned false.
```

C# 语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

请参阅

C# 参考

C# 编程指南

C# 运算符

|| 运算符（C# 参考）

条件或运算符 (||) 执行的逻辑或其 **bool** 操作数。如果第一个操作数计算结果为 **true**，第二个操作数对象不会计算。如果第一个操作数计算结果为 **false**，第二个运算符确定或表达式整体是否计算结果为 **true** 或 **false**。

备注

操作

```
x || y
```

对应于操作

```
x | y
```

但，如果 x 是 **true**，y 不会计算无论 y，的值，因为或操作是 **true**。此概念称为“短路计算”。

条件或运算符无法重载，但是，公共逻辑运算符和 真 和 假 运算符的重载，有一些限制的，也将条件逻辑运算符的重载。

在下面的示例中，表达式中使用的 **||** 计算只有第一个操作数。使用的表达式**|**计算两个操作数。在第二个示例中，因此，如果两个操作数计算，则运行时会发生异常。

```
class ConditionalOr
{
    // Method1 returns true.
    static bool Method1()
    {
        Console.WriteLine("Method1 called.");
        return true;
    }

    // Method2 returns false.
    static bool Method2()
    {
        Console.WriteLine("Method2 called.");
        return false;
    }

    static bool Divisible(int number, int divisor)
    {
```

```
        // If the OR expression uses ||, the division is not attemp
        // when the divisor equals 0.
        return !(divisor == 0 || number % divisor != 0);

        // If the OR expression uses |, the division is attempted w
        // the divisor equals 0, and causes a divide-by-zero except
        // Replace the return statement with the following line to
        // see the exception.
        //return !(divisor == 0 | number % divisor != 0);
    }

    static void Main()
    {
        // Example #1 uses Method1 and Method2 to demonstrate
        // short-circuit evaluation.

        Console.WriteLine("Regular OR:");
        // The | operator evaluates both operands, even though afte
        // Method1 returns true, you know that the OR expression is
        // true.
        Console.WriteLine("Result is {0}.\n", Method1() | Method2()

        Console.WriteLine("Short-circuit OR:");
        // Method2 is not called, because Method1 returns true.
        Console.WriteLine("Result is {0}.\n", Method1() || Method2(

        // In Example #2, method Divisible returns True if the
        // first argument is evenly divisible by the second, and Fa
        // otherwise. Using the | operator instead of the || operat
        // causes a divide-by-zero exception.

        // The following line displays True, because 42 is evenly
        // divisible by 7.
        Console.WriteLine("Divisible returns {0}.", Divisible(42, 7

        // The following line displays False, because 42 is not eve
        // divisible by 5.
        Console.WriteLine("Divisible returns {0}.", Divisible(42, 5

        // The following line displays False when method Divisible
        // uses ||, because you cannot divide by 0.
        // If method Divisible uses | instead of ||, this line
        // causes an exception.
        Console.WriteLine("Divisible returns {0}.", Divisible(42, 0
    }
}
/*
Output:
Regular OR:
Method1 called.
Method2 called.
Result is True.
```

```
Short-circuit OR:
Method1 called.
Result is True.

Divisible returns True.
Divisible returns False.
Divisible returns False.
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

<< 运算符（C# 参考）

左移运算符 (<<) 将第一个操作数向左移动第二个操作数指定的位数。第二个操作数的类型必须是一个 int 或具有向 **int** 的预定义隐式数值转换的类型。

备注

如果第一个操作数是 int 或 uint（32 位数），则移位数由第二个操作数的低 5 位给出。也就是实际的 shift 计数为 0 到 31 位。

如果第一个操作数是 long 或 ulong（64 位数），则移位数由第二个操作数的低 6 位给出。也就是实际的 shift 计数为 0 到 63 位。

不在移位后第一个操作数类型范围内的任意高序位均不会使用，低序空位用零填充。移位操作从不导致溢出。

用户定义的类型可重载 **<<** 运算符（请参见操作数）；第一个操作数的类型必须为用户定义的类型，第二个操作数的类型必须为 **int**。重载二元运算符时，也会隐式重载相应的赋值运算符（如果有）。

```
class MainClass11
{
    static void Main()
    {
        int i = 1;
        long lg = 1;
        // Shift i one bit to the left. The result is 2.
        Console.WriteLine("0x{0:x}", i << 1);
        // In binary, 33 is 100001\. Because the value of the five
        // bits is 1, the result of the shift is again 2\.
        Console.WriteLine("0x{0:x}", i << 33);
        // Because the type of lg is long, the shift is the value c
        // low-order bits. In this example, the shift is 33, and tl
        // lg is shifted 33 bits to the left.
        //     In binary:     10 0000 0000 0000 0000 0000 0000 000(
        //     In hexadecimal: 2    0    0    0    0    0    0    (
        Console.WriteLine("0x{0:x}", lg << 33);
    }
}
/*
Output:
0x2
0x2
0x200000000
*/
```

注释

请注意，i<<1 和 i<<33 给出的结果相同，因为 1 和 33 的低序 5 位相同。

请参阅

C# 参考

C# 编程指南

C# 运算符

>> 运算符（C# 参考）

右移运算符 (>>) 将第一个操作数向右移动第二个操作数所指定的位数。

备注

如果第一个操作数为 int 或 uint（32 位数），则移位数由第二个操作数的低五位给出（第二个操作数 & 0x1f）。

如果第一个操作数为 long 或 ulong（64 位数），则移位数由第二个操作数的低六位给出（第二个操作数 & 0x3f）。

如果第一个操作数为 int 或 long，则右移位是算术移位（高序空位设置为符号位）。如果第一个操作数为 uint 或 ulong 类型，则右移位是逻辑移位（高位填充 0）。

用户定义的类型可重载 **>>** 运算符；第一个操作数的类型必须为用户定义的类型，第二个操作数的类型必须为 int。有关更多信息，请参见 operator。重载二元运算符时，也会隐式重载相应的赋值运算符（如果有）。

```
class RightShift
{
    static void Main()
    {
        int i = -1000;
        Console.WriteLine(i >> 3);
    }
}
/*
Output:
-125
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

== 运算符（C# 参考）

对于预定义的值类型，如果操作数的值相等，则相等运算符 (==) 返回 true，否则返回 **false**。对于 string 以外的引用类型，如果两个操作数引用同一个对象，则 **==** 返回 **true**。对于 **string** 类型，**==** 比较字符串的值。

备注

用户定义的值类型可重载 **==** 运算符（请参见 operator）。用户定义的引用类型也可重载 == 运算符，尽管在默认情况下，无论对于预定义的引用类型还是用户定义的引用类型，**==** 的行为都与上面描述的相同。如果重载 **==**，则还必须重载 !=。 对于整数类型适用的运算对枚举类型通常也适用。

```
class Equality
{
    static void Main()
    {
        // Numeric equality: True
        Console.WriteLine((2 + 2) == 4);

        // Reference equality: different objects,
        // same boxed value: False.
        object s = 1;
        object t = 1;
        Console.WriteLine(s == t);

        // Define some strings:
        string a = "hello";
        string b = String.Copy(a);
        string c = "hello";

        // Compare string values of a constant and an instance: Tru
        Console.WriteLine(a == b);

        // Compare string references;
        // a is a constant but b is an instance: False.
        Console.WriteLine((object)a == (object)b);

        // Compare string references, both constants
        // have the same value, so string interning
        // points to same reference: True.
        Console.WriteLine((object)a == (object)c);
    }
}
/*
Output:
True
False
True
False
True
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

!= 运算符（C# 参考）

如果操作数相等，则不等运算符 (!=) 返回 false，否则，返回 true。为所有类型（包括字符串和对象）预定义了不等运算符。用户定义的类型可重载 **!=** 运算符。

备注

对于预定义的值类型，如果操作数的值不同，则不等运算符 (!=) 返回 true，否则，返回 false。对于 **string** 以外的引用类型，如果两个操作数引用不同的对象，则 **!=** 返回 true。对于 **string** 类型，**!=** 比较字符串的值。

用户定义的值类型可重载 **!=** 运算符（请参见 operator）。用户定义的引用类型也可重载 != 运算符，尽管在默认情况下，无论对于预定义的引用类型还是用户定义的引用类型，**!=** 的行为都与上面描述的相同。如果重载 **!=**，则还必须重载 ==。对于整数类型适用的运算对枚举类型通常也适用。

```
class InequalityTest
{
    static void Main()
    {
        // Numeric inequality:
        Console.WriteLine((2 + 2) != 4);

        // Reference equality: two objects, same boxed value
        object s = 1;
        object t = 1;
        Console.WriteLine(s != t);

        // String equality: same string value, same string objects
        string a = "hello";
        string b = "hello";

        // compare string values
        Console.WriteLine(a != b);

        // compare string references
        Console.WriteLine((object)a != (object)b);
    }
}
/*
Output:
False
True
False
False
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

<= 运算符（C# 参考）

所有数值和枚举类型都定义了“小于等于”关系运算符 (**<=**)，如果第一个操作数小于或等于第二个操作数，则该运算符将返回 **true**，否则返回 **false**。

备注

用户定义的类型可重载 **<=**运算符。有关更多信息，请参见 operator。如果重载 **<=**，则还必须重载 >=。 对于整数类型适用的运算对枚举类型通常也适用。

```
class LTE
{
    static void Main()
    {
        Console.WriteLine(1 <= 1.1);
        Console.WriteLine(1.1 <= 1.1);
    }
}
/*
Output:
True
True
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

explicit（C# 参考）

>= 运算符（C# 参考）

所有数值类型和枚举类型都定义“大于等于”关系运算符 **>=**，如果第一个操作数大于或等于第二个操作数，该运算符将返回 **true**，否则返回 **false**。

备注

用户定义的类型可重载 **>=**运算符。有关更多信息，请参见 operator。如果重载 **>=**，则还必须重载 <=。 对于整数类型适用的运算对枚举类型通常也适用。

```
class GTE
{
    static void Main()
    {
        Console.WriteLine(1.1 >= 1);
        Console.WriteLine(1.1 >= 1.1);
    }
}
/*
Output:
True
True
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

+= 运算符（C# 参考）

加法赋值运算符。

备注

使用 **+=** 赋值运算符的表达式，如

```
x += y
```

等效于

```
x = x + y
```

不同的是 x 只计算一次。 + 运算符的含义取决于 x 和 y 的类型（对于数值操作数，其含义为相加；对于字符串操作数，其含义为串联，等等）。

不能直接重载 **+=** 运算符，但用户定义的类型可重载 + 运算符（请参见 operator）。

+= 运算符还用于指定响应事件时要调用的方法；这类方法称为事件处理程序。在此上下文中使用 **+=** 运算符称为“订阅事件”。有关更多信息，请参见如何：订阅和取消订阅事件（C# 编程指南）。和委托（C# 编程指南）。

```
class AddAssigment
{
    static void Main()
    {
        //addition
        int a = 5;
        a += 6;
        Console.WriteLine(a);

        //string concatenation
        string s = "Hello";
        s += " world.";
        Console.WriteLine(s);
    }
}
/*
Output:
11
Hello world
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

-= 运算符（C# 参考）

减法赋值运算符。

备注

使用 **-=** 赋值运算符的表达式，如

```
x -= y
```

等效于

```
x = x - y
```

不同的是 x 只计算一次。 - 运算符的含义取决于 x 和 y 的类型（例如，对于数值操作数，其含义为相减；对于委托操作数，其含义为移除）。

不能直接重载 **-=** 运算符，但用户定义的类型可重载 - 运算符（请参见 operator）。

C# 中也使用 -= 运算符来取消订阅事件。有关更多信息，请参见如何：订阅和取消订阅事件（C# 编程指南）。

```
class MainClass3
{
    static void Main()
    {
        int a = 5;
        a -= 6;
        Console.WriteLine(a);
    }
}
/*
Output:
-1
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

*= 运算符（C# 参考）

二元乘法赋值运算符。

备注

使用 ***=** 赋值运算符的表达式，如

```
x *= y
```

等效于

```
x = x * y
```

不同的是 x 只计算一次。为数值类型预定义了 * 运算符以执行乘法操作。

不能直接重载 ***=** 运算符，但用户定义的类型可重载 * 运算符（请参见 operator）。

```
class MainClass10
{
    static void Main()
    {
        int a = 5;
        a *= 6;
        Console.WriteLine(a);
    }
}
/*
Output:
30
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

/= 运算符（C# 参考）

除法赋值运算符。

备注

使用 **/=** 赋值运算符的表达式，如

```
x /= y
```

等效于

```
x = x / y
```

不同的是 x 只计算一次。为数值类型预定义了 / 运算符以执行除法操作。

不能直接重载 **/=** 运算符，但用户定义的类型可重载 / 运算符（请参见 operator）。对于所有复合赋值运算符，隐式重载二元运算符会重载等效的复合赋值。

```
class MainClass2
{
    static void Main()
    {
        int a = 5;
        a /= 6;
        Console.WriteLine(a);
        double b = 5;
        b /= 6;
        Console.WriteLine(b);
    }
}
/*
Output:
0
0.833333333333333
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

%= 运算符（C# 参考）

其余部分赋值运算符。

备注

使用 **%=** 赋值运算符的表达式，例如

```
x %= y
```

等效于

```
x = x % y
```

，但 x 一次只计算。 % 运算符 预定义为数值类型可以在部门后计算其余部分。

不能直接重载 **%=** 运算符，但是，用户定义的类型能重载 % 运算符 (请参见 运算符（C# 参考）)。

```
class Test2
{
    static void Main()
    {
        int a = 5;
        a %= 3;
        Console.WriteLine(a);
    }
}
// Output: 2
```

请参阅

C# 参考

C# 编程指南

C# 运算符

&= 运算符（C# 参考）

“与”赋值运算符。

备注

使用 **&=** 赋值运算符的表达式，例如

```
x &= y
```

等效于

```
x = x & y
```

不同的是 x 只计算一次。 & 运算符对整数操作数执行按位逻辑“与”运算，对 **bool** 操作数执行逻辑“与”运算。

不能直接重载 **&=** 运算符，但用户定义的类型可重载二元 & 运算符（请参见 operator）。

```
class AndAssignment
{
    static void Main()
    {
        int a = 0x0c;
        a &= 0x06;
        Console.WriteLine("0x{0:x8}", a);
        bool b = true;
        b &= false;
        Console.WriteLine(b);
    }
}
/*
Output:
0x00000004
False
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

|= 运算符（C# 参考）

“或”赋值运算符。

备注

使用 **|=** 赋值运算符的表达式，例如

```
x |= y
```

等效于

```
x = x | y
```

不同的是 x 只计算一次。 | 运算符对整型操作数执行按位逻辑“或”运算，对布尔操作数执行逻辑“或”运算。

不能直接重载 **|=** 运算符，但用户定义的类型可以重载 | 运算符（请参见 operator）。

```
class MainClass7
{
    static void Main()
    {
        int a = 0x0c;
        a |= 0x06;
        Console.WriteLine("0x{0:x8}", a);
        bool b = true;
        b |= false;
        Console.WriteLine(b);
    }
}
/*
Output:
0x0000000e
True
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

^= 运算符（C# 参考）

“异或”赋值运算符。

备注

下列形式的表达式

```
x ^= y
```

按如下规则计算：

```
x = x ^ y
```

不同的是 x 只计算一次。 ^ 运算符对整数操作数执行按位“异或”运算，对 bool 操作数执行逻辑“异或”运算。

不能直接重载 ^= 运算符，但用户定义的类型可重载 ^ operator（请参见 operator）。

```
class XORAssignment
{
    static void Main()
    {
        int a = 0x0c;
        a ^= 0x06;
        Console.WriteLine("0x{0:x8}", a);
        bool b = true;
        b ^= false;
        Console.WriteLine(b);
    }
}
/*
Output:
0x0000000a
True
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

<<= 运算符（C# 参考）

左移赋值运算符。

备注

下列形式的表达式

```
x <<= y
```

按如下规则计算：

```
x = x << y
```

不同的是 x 只计算一次。 << 运算符将 x 向左移动 y 指定的位数。

不能直接重载 **<<=** 运算符，但用户定义的类型可重载 << 运算符（请参见 operator）。

```
class MainClass9
{
    static void Main()
    {
        int a = 1000;
        a <<= 4;
        Console.WriteLine(a);
    }
}
/*
Output:
16000
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

>>= 运算符（C# 参考）

右移赋值运算符。

备注

下列形式的表达式

```
x >>= y
```

按如下规则计算：

```
x = x >> y
```

不同的是 x 只计算一次。 >> 运算符根据 y 指定的量对 x 进行右移位。

不能直接重载 >>= 运算符，但用户定义的类型可重载 >> 运算符（请参见 operator）。

示例

```
class MainClass8
{
    static void Main()
    {
        int a = 1000;
        a >>= 4;
        Console.WriteLine(a);
    }
}
/*
Output:
62
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

-> 运算符（C# 参考）

-> 运算符将指针取消引用与成员访问组合在一起。

备注

以下形式的表达式

```
x->y
```

（其中 x 为 T* 类型的指针，y 为 T 的成员）等效于

```
(*x).y
```

只能在标记为不安全的代码中使用 **->** 运算符。

不能重载 **->** 运算符。

```
// compile with: /unsafe

struct Point
{
    public int x, y;
}

class MainClass12
{
    unsafe static void Main()
    {
        Point pt = new Point();
        Point* pp = &pt;
        pp->x = 123;
        pp->y = 456;
        Console.WriteLine("{0} {1}", pt.x, pt.y);
    }
}
/*
Output:
123 456
*/
```

请参阅

C# 参考

C# 编程指南

C# 运算符

?? 运算符（C# 参考）

?? 运算符称作 null 合并运算符。如果此运算符的左操作数不为 null，则此运算符将返回左操作数；否则返回右操作数。

备注

可以为 null 的类型可以表示类型的域中的值，或者值可以是未定义的（在这种情况下，值为 null）。当左操作数具有一个值为 null 的可以为 null 的类型时，可以使用 **??** 运算符的语法表现力来返回适当的值（右操作数）。如果在尝试将可以为 null 值的类型分配给不可以为 null 值的类型时没有使用 **??** 运算符，则会生成编译时错误。如果使用强制转换，且当前未定义可以为 null 值的类型，则会引发 **InvalidOperationException** 异常。

有关详细信息，请参阅可以为 null 的类型（C# 编程指南）。

即使 ?? 运算符的两个参数都是常量，也不能将其结果视为常量。

```
class NullCoalesce
{
    static int? GetNullableInt()
    {
        return null;
    }

    static string GetStringValue()
    {
        return null;
    }

    static void Main()
    {
        int? x = null;

        // Set y to the value of x if x is NOT null; otherwise,
        // if x = null, set y to -1.
        int y = x ?? -1;

        // Assign i to return value of the method if the method's r
        // is NOT null; otherwise, if the result is null, set i to
        // default value of int.
        int i = GetNullableInt() ?? default(int);

        string s = GetStringValue();
        // Display the value of s if s is NOT null; otherwise,
        // display the string "Unspecified".
        Console.WriteLine(s ?? "Unspecified");
    }
}
```

请参阅

C# 参考

C# 编程指南

C# 运算符

可以为 null 的类型（C# 编程指南）

“提升”的准确含义是什么？

=> 运算符（C# 参考）

=> 标记称作 lambda 运算符。该标记在 lambda 表达式中用来将左侧的输入变量与右侧的 lambda 体分离。Lambda 表达式是与匿名方法类似的内联表达式，但更加灵活；在以方法语法表示的 LINQ 查询中广泛使用了 Lambda 表达式。有关更多信息，请参见Lambda 表达式（C# 编程指南）。

下面的示例演示两种查找并显示最短的字符串的长度在字符数组中的字符串。该示例的第一部分将 lambda 表达式 (w => w.Length) words 于数组的每个元素都使用 Min<TSource> 方法查找最小长度。为了进行比较，该示例的第二部分演示一个较长的解决方案使用查询语法执行相同操作。

```
string[] words = { "cherry", "apple", "blueberry" };

// Use method syntax to apply a lambda expression to each element
// of the words array.
int shortestWordLength = words.Min(w => w.Length);
Console.WriteLine(shortestWordLength);

// Compare the following code that uses query syntax.
// Get the lengths of each word in the words array.
var query = from w in words
            select w.Length;
// Apply the Min method to execute the query and get the shortest l
int shortestWordLength2 = query.Min();
Console.WriteLine(shortestWordLength2);

// Output:
// 5
// 5
```

备注

=> 运算符具有与赋值运算符 (=) 相同的优先级，并且是右结合运算符。

可以显式指定输入变量的类型或让编译器推断类型；仍，该变量是强类型在编译时。当指定类型时，必须将该类型括号中的名称和变量名，如下例所示。

```
int shortestWordLength = words.Min((string w) => w.Length);
```

下面的代码演示如何编写采用两个参数标准查询运算符 Enumerable.Where<TSource> 的超加载的 lambda 表达式。由于 lambda 表达式具有多个参数，必须括在括号中的参数。第二个参数，index，表示当前元素的索引集合中的。 **Where** 表达式返回长度小于其在数组的索引位置小于的任何字符串。

```
static void Main(string[] args)
{
    string[] digits = { "zero", "one", "two", "three", "four", "fiv
            "six", "seven", "eight", "nine" };

    Console.WriteLine("Example that uses a lambda expression:");
    var shortDigits = digits.Where((digit, index) => digit.Length <
    foreach (var sD in shortDigits)
    {
        Console.WriteLine(sD);
    }

    // Compare the following code, which arrives at the same list o
    // digits but takes more work to get there.
    Console.WriteLine("\nExample that uses a for loop:");
    List<string> shortDigits2 = new List<string>();
    for (var i = 0; i < digits.Length; i++)
    {
        if (digits[i].Length < i)
            shortDigits2.Add(digits[i]);
    }

    foreach (var d in shortDigits2)
    {
        Console.WriteLine(d);
    }
    // Output:
    // Example that uses a lambda expression:
    // five
    // six
    // seven
    // eight
    // nine

    // Example that uses a for loop:
    // five
    // six
    // seven
    // eight
    // nine
}
```

请参阅

C# 参考

C# 编程指南

Lambda 表达式（C# 编程指南）

NULL 条件运算符（C# 和 Visual Basic）

用于在执行成员访问 (**?.**) 或索引 (**?[**) 操作之前，测试是否存在 NULL。这些运算符可帮助编写更少的代码来处理 null 检查，尤其是对于下降到数据结构。

VB

```
int? length = customers?.Length; // null if customers is null
Customer first = customers?[0];  // null if customers is null
int? count = customers?[0]?.Orders?.Count();  // null if customers,
```

最后一个示例演示 NULL 条件运算符会短路。如果条件成员访问和索引操作链中的某个操作返回 NULL，则该链其余部分的执行将停止。表达式中优先级较低的其他操作将继续。例如，以下的示例中的 E 将始终执行，**??** 和 **==** 操作将执行。

visualbasicANDcsharp

```
A?.B?.C?[0] ?? E
A?.B?.C?[0] == E
```

NULL 条件成员访问的另一个用途是使用非常少的代码以线程安全的方式调用委托。旧方法需要如下所示的代码：

VB

```
var handler = this.PropertyChanged;
if (handler != null)
    handler(…)
```

新的方法是要简单得多：

visualbasicANDcsharp

```
PropertyChanged?.Invoke(e)
```

新方法是线程安全的，因为编译器生成代码以评估 PropertyChanged（仅一次），从而使结果保持在临时变量中。

你需要显式调用 Invoke 方法，因为不存在 NULL 条件委托调用语法 PropertyChanged?(e)。有太多不明确的分析情况来允许它。

语言规范

有关详细信息，请参阅 C# 语言规范。该语言规范是 C# 语法和用法的权威资料。

有关详细信息，请参阅 Visual Basic 语言参考。

请参阅

C# 参考

C# 编程指南

Visual Basic 语言参考

Visual Basic 编程指南

C# 预处理器指令

本节包含有关以下 C# 预处理器指令的信息。

#if

#else

#elif

#endif

define

#undef

#warning

#error

#line

#region

#endregion

#pragma

#pragma warning

#pragma checksum

请参见各个主题有关更多信息和示例。

虽然编译器没有单独的预处理器，本节中介绍的处理指令，就象一个。它们在条件编译用于帮助。与 C 和 C++ 指令不同，不能使用这些指令创建宏。

预处理器指令必须是行上的唯一指令。

请参阅

C# 参考

C# 编程指南

#if（C# 参考）

如果 C# 编译器遇到最后面跟有 #endif 指令的 **#if** 指令，则仅当指定的符号已定义时，它才会编译这两个指令之间的代码。与 C 和 C++ 不同，您不能对符号赋予数值；C# 中的 #if 语句是 Boolean，仅测试符号是否已定义。例如，

```
#define DEBUG
// ...
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

使用运算符 ==（相等）和 !=（不相等）只能测试出结果为 true 还是 false。True 表示符号已定义。语句 #if DEBUG 与 #if (DEBUG == true) 的含义相同。可以使用运算符&& (and)、|| (or) 和 !(无) 计算多个符号是否定义了。还可以用括号将符号和运算符分组。

备注

结合使用 **#if** 与 #else、#elif、#endif、#define 和 #undef 指令，可以根据一个或多个符号是否存在来包含或排除代码。在编译调试版本的代码或针对特定配置进行编译时，这会很有用。

以 **#if** 指令开始的条件指令必须用 **#endif** 指令显式终止。

#define 使您可以定义一个符号，通过将该符号用作传递给 #if 指令的表达式，使该表达式计算为 **true**。

也可以用 /define 编译器选项来定义符号。可以用 #undef 来取消定义符号。

用 **/define** 或 **#define** 定义的符号与具有同一名称的变量不冲突。即，不应将变量名传递到预处理器指令，并且只能用预处理器指令计算符号。

用 **#define** 创建的符号的范围是在其中定义该符号的文件。

```
// preprocessor_if.cs
#define DEBUG
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
#if (DEBUG && !MYTEST)
        Console.WriteLine("DEBUG is defined");
#elif (!DEBUG && MYTEST)
        Console.WriteLine("MYTEST is defined");
#elif (DEBUG && MYTEST)
        Console.WriteLine("DEBUG and MYTEST are defined");
#else
        Console.WriteLine("DEBUG and MYTEST are not defined");
#endif
    }
}
```

```
定义 DEBUG 和 MYTEST
```

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

#else（C# 参考）

#else 允许您创建复合条件指令，因此，如果前面的 #if 或（可选）#elif 指令中的任何表达式都不为 **true**，则编译器将计算 **#else** 与后面的 **#endif** 之间的所有代码。

备注

#endif 必须是 **#else** 之后的下一条预处理器指令。有关如何使用 **#else** 的示例，请参见 #if。

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

#elif（C# 参考）

#elif 使您得以创建复合条件指令。如果之前的#if（C# 参考） 和任何之前的，可选的，**#elif** 关系表达式的值为**true**，则**#elif**表达式将被执行。如果 **#elif** 表达式计算为 **true**，编译器将计算位于 **#elif** 和下一个条件指令之间的所有代码。例如：

```
#define VC7
//...
#if debug
    Console.Writeline("Debug build");
#elif VC7
    Console.Writeline("Visual Studio 7");
#endif
```

可以使用运算符 **==**（相等）、**!=**（不相等）、**&&**（与）及 **||**（或）来计算多个符号。还可以用括号将符号和运算符分组。

备注

#elif 等效于使用：

```
#else
#if
```

使用 **#elif** 更简单，因为每个 **#if** 都需要一个 #endif（C# 参考），而 **#elif** 即使在没有匹配的 **#endif** 时也可以使用。

有关如何使用 **#elif** 的示例，请参见 #if（C# 参考）。

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

#endif（C# 参考）

#endif 指定以 #if 指令开头的条件指令的结尾。例如，

```
      #define DEBUG
// ...
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

备注

以 **#if** 指令开始的条件指令必须用 **#endif** 指令显式终止。有关如何使用 **#endif** 的示例，请参见 #if（C# 参考）。

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

#define（C# 参考）

使用 **#define** 定义符号。当您将符号用作传递给 #if 指令的表达式时，此表达式的计算结果为 **true**，如下例所示：

define DEBUG

备注

注意
不能像在 C 和 C++ 中的通常做法一样，使用 **#define** 指令来声明常数值。最好是将 C# 中的常数定义为类或结构的静态成员。如果具有多个像这样的常数，可以考虑创建一个单独的“Constants”类来保存这些常数。

符号可用于指定编译的条件。可以使用 #if 或 #elif 来测试符号。还可以使用 **conditional** 特性执行条件编译。

可以定义符号，但是无法对符号赋值。 **#define** 指令必须在使用任何不是预处理器指令的指令之前出现在文件中。

也可以用 /define 编译器选项来定义符号。可以用 #undef 来取消定义符号。

用 **/define** 或 **#define** 定义的符号与具有同一名称的变量不冲突。即，不应将变量名传递到预处理器指令，并且只能用预处理器指令计算符号。

用 **#define** 创建的符号范围是在其中定义该符号的文件。

如以下示例所示，您必须将 **#define** 指令置于文件的顶部。

```
#define DEBUG
//#define TRACE
#undef TRACE

using System;

public class TestDefine
{
    static void Main()
    {
#if (DEBUG)
        Console.WriteLine("Debugging is enabled.");
#endif

#if (TRACE)
     Console.WriteLine("Tracing is enabled.");
#endif
    }
}
// Output:
// Debugging is enabled.
```

有关如何取消定义符号的示例，请参见 #undef（C# 参考）。

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

const（C# 参考）

How to: Compile Conditionally with Trace and Debug

#undef（C# 参考）

#if（C# 参考）

#undef（C# 参考）

#undef 使您可以取消符号的定义，以便通过将该符号用作 #if 指令中的表达式，使表达式的计算结果为 **false**。

可以使用 #define 指令或 /define 编译器选项定义符号。在使用任何不是指令的语句之前，必须在文件中使用 **#undef** 指令。

```
// preprocessor_undef.cs
// compile with: /d:DEBUG
#undef DEBUG
using System;
class MyClass
{
    static void Main()
    {
#if DEBUG
        Console.WriteLine("DEBUG is defined");
#else
        Console.WriteLine("DEBUG is not defined");
#endif
    }
}
```

```
未定义 DEBUG
```

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

#warning（C# 参考）

#warning 使您得以从代码的特定位置生成一级警告。 例如：

```
#warning Deprecated code in this method.
```

备注

#warning 通常用在条件指令中。 也可以用 #error（C# 参考） 生成用户定义的错误。

示例

```
// preprocessor_warning.cs
// CS1030 expected
#define DEBUG
class MainClass
{
    static void Main()
    {
#if DEBUG
#warning DEBUG is defined
#endif
    }
}
```

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

#error（C# 参考）

#error 使您可以从代码中的特定位置生成错误。例如：

```
#error Deprecated code in this method.
```

备注

#error 通常用在条件指令中。

也可以用 #warning（C# 参考） 生成用户定义的警告。

```
// preprocessor_error.cs
// CS1029 expected
#define DEBUG
class MainClass
{
    static void Main()
    {
#if DEBUG
#error DEBUG is defined
#endif
    }
}
```

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

#line（C# 参考）

#line 使您可以修改编译器的行号以及（可选）错误和警告的文件名输出。下面的示例说明如何报告与行号关联的两个警告。 #line 200 指令将行号强制设置为 200（尽管默认行号为 #7），在执行下一条 #line 指令之前，文件名将报告为“Special”。#line default 指令将行号恢复为默认行号，默认行号对前一条指令重新编号的行进行计数。

```
class MainClass
{
    static void Main()
    {
#line 200 "Special"
        int i;    // CS0168 on line 200
        int j;    // CS0168 on line 201
#line default
        char c;   // CS0168 on line 9
        float f;  // CS0168 on line 10
#line hidden // numbering not affected
        string s;
        double d; // CS0168 on line 13
    }
}
```

备注

#line 指令可能由生成过程中的自动中间步骤使用。例如，如果行从原始的源代码文件中移除，但是您仍希望编译器基于文件中的原始行号生成输出，则可以移除行，然后用 **#line** 模拟原始行号。

#line hidden 指令对调试器隐藏若干连续的行，这样当开发人员在逐句通过代码时，将会跳过 **#line hidden** 和下一个 **#line** 指令（假定它不是另一个 **#line hidden** 指令）之间的所有行。此选项也可用来使 ASP.NET 能够区分用户定义的代码和计算机生成的代码。尽管 ASP.NET 是此功能的主要使用者，但很可能将有更多的源生成器使用它。

#line hidden 指令不会影响错误报告中的文件名或行号。即，如果在隐藏块中遇到错误，编译器将报告当前文件名和错误的行号。

#line filename 指令指定您希望出现在编译器输出中的文件名。默认情况下，使用源代码文件的实际名称。文件名必须用双引号 ("") 引起来且前面必须带一个行号。

源代码文件可以具有 **#line** 指令的任何编号。

示例 1

下面的示例说明调试器如何忽略代码中的隐藏行。运行此示例时，它将显示三行文本。但是，当设置如示例所示的断点并按 F10 键逐句通过代码时，您将看到调试器忽略了隐藏行。还请注意，即使在隐藏行上设置断点，调试器仍会忽略它。

```
// preprocessor_linehidden.cs
using System;
class MainClass
{
    static void Main()
    {
        Console.WriteLine("Normal line #1."); // Set break point he
#line hidden
        Console.WriteLine("Hidden line.");
#line default
        Console.WriteLine("Normal line #2.");
    }
}
```

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

#region（C# 参考）

#region 使您可以在使用 Visual Studio 代码编辑器的大纲显示功能时指定可展开或折叠的代码块。在较长的代码文件中，能够折叠或隐藏一个或多个区域会十分便利，这样，您可将精力集中于当前处理的文件部分。下面的示例演示如何定义区域：

```
      #region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

备注

#region 块必须以 #endregion 指令终止。

#region 块不能与 #if 块重叠。但是，可以将 **#region** 块嵌套在 **#if** 块内，或将 **#if** 块嵌套在 **#region** 块内。

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

#endregion（C# 参考）

#endregion 标记 #region 块的结尾。例如：

```
    #region MyClass definition
class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

#pragma（C# 参考）

#pragma 为编译器提供特殊的指令，以说明如何编译包含杂注的文件。这些指令必须是编译器支持的指令。也就是说，不能使用 **#pragma** 创建自定义预处理指令。Microsoft C# 编译器支持以下两个 **#pragma** 指令：

#pragma warning（C# 参考）

#pragma checksum（C# 参考）

语法

```
#pragma pragma-name pragma-arguments
```

参数

pragma-name

可识别杂注的名称。

pragma-arguments

特定于杂注的参数。

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

#pragma warning（C# 参考）

#pragma checksum（C# 参考）

#pragma warning（C# 参考）

#pragma warning 可启用或禁用某些警告。

语法

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

参数

warning-list

警告编号的逗号分隔列表。只输入数字，不包括前缀 "CS"。

当没有指定警告编号时，**disable** 禁用所有警告，而 **restore** 启用所有警告。

注意
要在 Visual Studio 中查找警告编号，请先生成项目，再在“输出”窗口中查找警告编号。

```
// pragma_warning.cs
using System;

#pragma warning disable 414, 3021
[CLSCompliant(false)]
public class C
{
    int i = 1;
    static void Main()
    {
    }
}
#pragma warning restore 3021
[CLSCompliant(false)]  // CS3021
public class D
{
    int i = 1;
    public static void F()
    {
    }
}
```

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

C# Compiler Errors

#pragma checksum（C# 参考）

生成源文件的校验和，以帮助调试 ASP.NET 页。

语法

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

参数

"filename"

要求监视更改或更新的文件的名称。

"{guid}"

文件的全局唯一标识符 (GUID)。

"checksum_bytes"

十六进制数的字符串，表示校验和的字节。必须是偶数位的十六进制数。奇数位的十六进制数字会导致编译时警告，然后指令被忽略。

备注

Visual Studio 调试器使用校验和来确保找到的总是正确的源。编译器计算源文件的校验和，然后将输出发出到程序数据库 (PDB) 文件。最后，调试器使用 PDB 来比较它为源文件计算的校验和。

此解决方案不适用于 ASP.NET 项目，因为算出的校验和是生成的源文件的校验和，而不是 .aspx 文件的校验和。为解决此问题，**#pragma checksum** 为 ASP.NET 页提供了校验和支持。

在 Visual C# 中创建 ASP.NET 项目时，生成的源文件包含 .aspx 文件（从该文件生成源文件）的校验和。然后，编译器将此信息写入 PDB 文件。

如果编译器在该文件中没有遇到 **#pragma checksum** 指令，它将计算校验和，然后将算出的值写入 PDB 文件。

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{3673e4ca-6098-4ec1-890f-8fceb2
    }
}
```

请参阅

C# 参考

C# 编程指南

C# 预处理器指令

C# Compiler Options

编译器生成可执行 (.exe) 文件、动态链接库 (.dll) 或者代码模块(.netmodule)。

每个编译器选项均以两种形式提供：**-option** 和 **/option**。本文档仅显示 **/option** 形式。

在 Visual Web Developer 2008 中，在 web.config 文件中设置编译器选项。有关详细信息，请参阅<compiler> 元素。

本节内容

Command-line Building With csc.exe

有关从命令行生成 Visual C# 应用程序的信息。

How to: Set Environment Variables for the Visual Studio Command Line

提供运行 vsvars32.bat 以启用命令行编译的步骤。

部署 C# 应用程序

描述用于部署 C# 应用程序的选项。

按类别列出的 C# 编译器选项

编译器选项的分类列表。

按字母顺序列出的 C# 编译器选项

按字母顺序排序的编译器选项列表。

相关章节

“项目设计器”->“生成”页

设置控制项目的编译、生成和调试方法的属性。包含有关在 Visual C# 项目中自定义生成步骤的信息。

默认与自定义生成

有关生成类型和配置的信息。

准备和管理生成

在 Visual Studio 开发环境内生成的过程。

Command-line Building With csc.exe

可以通过在命令行上键入 C# 编译器的名为 (csc.exe) 的可执行文件来调用 C# 编译器。

如果您使用 **Visual Studio** 命令提示符 窗口，所有必需的环境变量都将为您设置. 在 Windows 7 中，您可以通过打开 开始 菜单中的 Microsoft Visual Studio *Version*\Visual Studio Tools 文件夹来访问这个窗口.在 Windows 8 中， Visual Studio 命令行被命名为 开发者命令提示符 **for VS2012**，您可以在开始屏幕搜索到它.

如果使用标准命令提示符窗口，必须调整路径，然后才能从计算机的任何子目录调用 csc.exe。您还必须运行 vsvars32.bat 来设置适当的环境变量以支持命令行构建。有关 vsvars32.bat 内容的详细信息，包括如何找到它以及运行它，请参见 How to: Set Environment Variables for the Visual Studio Command Line.

如果您使用的计算机只安装有 Windows 软件开发包 (SDK)，那么您可在**“SDK** 命令提示符”上使用 C# 编译器。SDK 命令提示符可通过**“Microsoft .NET Framework SDK”**菜单选项打开

您也可以使用 MSBuild 通过编程方式生成 C# 程序。有关详细信息，请参阅MSBuild。

csc.exe 可执行文件通常位于系统目录下的 Microsoft.NET\Framework_Version_ 文件夹中。根据每台计算机上的实际配置，此位置可能有所不同。如果在您的计算机上安装了不止一个 .NET Framework 的版本，您将发现此文件的多个版本。有关此类安装的更多信息，请参见Determining Which Version of the .NET Framework Is Installed。

提示
当您使用 Visual Studio IDE 构建项目时，您可以在 输出 窗口显示 **csc** 命令以及与之关联的编译器选项.要显示更多的信息，依据指令 如何：查看、保存和配置生成日志文件 改变日志数据的冗余版本 -- 常规 或 详细.在您的项目重新生成之后，在 输出 窗口中查找 **csc** 以查看所调用的 C# 编译器.

主题内容

- 命令行语法规则
- 命令行示例
- C# 编译器和 C++ 编译器输出之间的差异

C# 编译器的命令行语法规则

在解释操作系统命令行上给出的参数时，C# 编译器使用下列规则：

- 参数用空白分隔，空白可以是一个空格或制表符。
- ^ 字符 (^) 未被识别为转义符或者分隔符。该字符在被传递给程序中的 argv 数组前，由操作系统的命令行分析器进行处理。
- 无论其中是否包含空白，带双引号的字符串 ("string") 均被解释为单个参数。带引号的字符串可以嵌入在参数内。
- 前面有反斜杠的双引号 (\") 被解释为原义双引号字符 (")。
- 反斜杠按其原义解释，除非它们紧位于双引号之前。
- 如果偶数个反斜杠后跟双引号，则每对反斜杠中有一个反斜杠放置在 argv 数组中，而双引号被解释为字符串分隔符。
- 如果奇数个反斜杠后跟双引号，则每对反斜杠中有一个反斜杠放置在 argv 数组中，而双引号由剩余那个反斜杠进行“转义”。这会将双引号字符 (") 添加到 argv 中。

C# 编译器的示例命令行

- 编译 File.cs 以产生 File.exe：

```
csc File.cs
```

- 编译 File.cs 以产生 File.dll：

```
csc /target:library File.cs
```

- 编译 File.cs 并创建 My.exe：

```
csc /out:My.exe File.cs
```

- 通过使用优化和定义 DEBUG 符号，编译当前目录中的所有 C# 文件。输出为 File2.exe：

```
csc /define:DEBUG /optimize /out:File2.exe *.cs
```

- 编译当前目录中的所有 C# 文件，以生成 File2.dll 的调试版本。不显示任何徽标和警告：

```
csc /target:library /out:File2.dll /warn:0 /nologo /debug *.cs
```

- 将当前目录中的所有 C# 文件编译为 Something.xyz（一个 DLL）：

```
csc /target:library /out:Something.xyz *.cs
```

C# 编译器和 C++ 编译器输出之间的差异

由于调用 C# 编译器的结果没有创建目标文件(.obj)；输出文件被直接创建。 因此，C# 编译器不需要链接器。

请参阅

C# Compiler Options

C# Compiler Options Listed Alphabetically

C# Compiler Options Listed by Category

Main() 和命令行参数（C# 编程指南）

命令行参数（C# 编程指南）

如何：显示命令行参数（C# 编程指南）

如何：使用 foreach 访问命令行参数（C# 编程指南）

Main() 返回值（C# 编程指南）

How to: Set Environment Variables for the Visual Studio Command Line

vsvars32.bat 文件设置适当的环境变量以启用命令行生成。有关 vsvars32.bat 的详细信息，请参阅知识库文章 Q248802。

如果安装了当前版本 Visual Studio 的计算机上也安装了 Visual Studio 的早期版本，则你不应该在同一命令提示符窗口中运行不同版本的 vsvars32.bat 或 vcvars32.bat。

若要运行 VSVARS32.BAT

1. 从"开始"菜单，打开"VS2012 开发者命令提示"。
2. 更改为安装目录的 Program Files\Microsoft Visual Studio *Version*\Common7\Tools 或 Program Files (x86)\Microsoft Visual Studio *Version*\Common7\Tools 子目录。
3. 通过键入 VSVARS32 运行 VSVARS32.bat。

⚠小心
VSVARS32.bat 在各个计算机间可能大不相同。不要使用另一台计算机中的 VSVARS32.bat 替换丢失或损坏的 VSVARS32.bat 文件。而是应重新运行安装程序以替换丢失的文件。

请参阅

Command-line Building With csc.exe

Deployment of C# Applications

当您完成生成您的 C# 应用程序之后，下一步是分发 C# 应用程序。C# 是一种 .NET 语言；因此，将任何 C# 可执行文件分发到其他计算机上要求在每台执行计算机上安装 .NET Framework（可能还包括特定于您的应用程序的其他依赖项）。您有各种可用于分发 .NET Framework 的选项。有关概述，请参见针对开发人员的部署指南。

将完成的应用程序移动到其他计算机上的过程通常称为“部署”。Microsoft 的开发环境提供了部署机制；有关更多信息，请参见部署应用程序和组件。

如果您主要是从命令行生成和分发，您可能需要考虑其他部署和再分发依赖项的方法。

请参阅

Command-line Building With csc.exe

C# Compiler Options Listed by Category

下列编译器选项按类别排序。关于按字母顺序排列的列表，请参阅按字母顺序列出的 C# 编译器选项。

优化

选项	目标
/filealign	指定输出文件中各节的大小。
/optimize	启用/禁用优化。

输出文件

选项	目标
/doc	指定要将已处理的文档注释写入到的 XML 文件。
/out	指定输出文件。
/pdb	指定 .pdb 文件的文件名和位置。
/platform	指定输出平台。
/preferreduilang	指定编译器输出的语言。
/target	使用下列五个选项之一指定输出文件的格式：/target:appcontainerexe、/target:exe、/target:library、/ta 或 /target:winmdobj。
/modulename:<string>	指定源模块的名称

.NET Framework 程序集

选项	目标
/addmodule	指定一个或多个模块作为此程序集的一部分。
/delaysign	指示编译器添加公钥，但将此程序集保留为未签名状态。
/keycontainer	指定加密密钥容器的名称。
/keyfile	指定包含加密密钥的文件名。
/lib	指定通过 /reference 的方式引用的程序集的位置。
/nostdlib	指示编译器不导入标准库 (mscorlib.dll)。
/reference	从包含程序集的文件导入元数据。
/analyzer	从此程序集（缩写形式：/a）运行分析器
/additionalfile	命名其他文件，这些文件不会直接影响代码生成，但可能由分析器用于生成错误或警告。

调试/错误检查

选项	目标
/bugreport	创建一个文件，其中包含可以轻松报告 bug 的信息。
/checked	指定溢出数据类型边界的整数算法是否将导致运行时异常。
/debug	指示编译器发出调试信息。
/errorreport	设置错误报告行为。
/fullpaths	指定编译器输出中文件的绝对路径。
/nowarn	取消编译器对指定警告的生成。
/warn	设置警告等级。
/warnaserror	将警告提升为错误。
/ruleset:<file>	指定可禁用特定诊断的规则集文件。

预处理器

选项	目标
/define	定义预处理器符号。

资源

选项	目标
/link	使指定程序集中的 COM 类型信息对项目可用。
/linkresource	创建指向托管资源的链接。
/resource	将 .NET Framework 资源嵌入到输出文件。
/win32icon	指定要插入到输出文件的 .ico 文件。
/win32res	指定要插入到输出文件的 Win32 资源。

杂项

选项	目标
@	指定响应文件。
/?	列出到 stdout 的编译器选项。
/baseaddress	指定要加载 DLL 的首选基址。
/codepage	指定要用于编译中所有源代码文件的代码页。
/help	列出到 stdout 的编译器选项。
/highentropyva	指定可执行文件支持地址空间布局随机化 (ASLR)。
/langversion	指定语言版本模式：ISO-1、ISO-2、3、4、5、6 或默认
/main	指定 **Main** 方法的位置。
/noconfig	指示编译器不使用 csc.rsp 进行编译。
/nologo	禁止显示编译器横幅信息。
/recurse	搜索要编译的源文件的子目录。
/subsystemversion	指定可执行文件可以使用的子系统的最低版本。
/unsafe	启用使用 unsafe 关键字的代码编译。
/utf8output	显示使用 UTF-8 编码的编译器输出。
/parallel[+|-]	指定是否使用并发生成 (+)。
/checksumalgorithm:<alg>	指定用于计算 PDB 中存储的源文件校验和的算法。支持的值为：SHA1（默认值）或 SHA256。

已过时的选项

| **/incremental** | 启用增量编译。 |

请参阅

C# Compiler Options

C# Compiler Options Listed Alphabetically

How to: Set Environment Variables for the Visual Studio Command Line

C# Compiler Options Listed Alphabetically

下列编译器选项按字母顺序排序。有关按类别排序的列表，请参阅C# Compiler Options Listed by Category。

选项	目
@	有关更多选项，请阅读响应文件。
/?	向 stdout 显示用法消息。
/additionalfile	命名其他文件，这些文件不会直接影响代码生成，但可
/addmodule	将指定的模块链接到此程序集中
/analyzer	从此程序集（缩写形式：/a）运行分析器
/appconfig	在程序集绑定时指定 app.config 的位置。
/baseaddress	指定要生成的库的基址。
/bugreport	创建“Bug 报告”文件。如果与 **/errorreport:prompt** 或 息一起发送。
/checked	使编译器生成溢出检查。
/checksumalgorithm:<alg>	指定用于计算 PDB 中存储的源文件校验和的算法。支
/codepage	指定在打开源文件时使用的代码页。
/debug	发出调试信息。
/define	定义条件编译符号。
/delaysign	仅使用强名称密钥公共部分对程序集进行延迟签名。
/doc	指定要生成的 XML 文档文件。
/errorreport	指定如何处理内部编译器错误：prompt、send 或 none
/filealign	指定用于输出文件节的对齐方式。
/fullpaths	使编译器生成完全限定的路径。
/help	向 stdout 显示用法消息。
/highentropyva	指定支持高熵 ASLR。
/incremental	启用增量编译 [已过时]。
/keycontainer	指定强名称密钥容器。
/keyfile	指定强名称密钥文件。

/langversion:<string>	指定语言版本模式：ISO-1、ISO-2、3、4、5、6 或默
/lib	指定要在其中搜索引用的附加目录。
/link	使指定程序集中的 COM 类型信息对项目可用。
/linkresource	将指定的资源链接到此程序集。
/main	指定包含入口点的类型（忽略所有其他可能的入口点）
/moduleassemblyname	指定 .netmodule 可以访问非公共类型的程序集。
/modulename:<string>	指定源模块的名称
/noconfig	指示编译器不自动包含 CSC.RSP 文件。
/nologo	取消编译器版权消息。
/nostdlib	指示编译器不引用标准库 (mscorlib.dll)。
/nowarn	禁用特定的警告消息
/nowin32manifest	指示编译器不在可执行文件中嵌入应用程序清单。
/optimize	启用/禁用优化。
/out	指定输出文件名（默认值：包含主类的文件或第一个文
/parallel[+|-]	指定是否使用并发生成 (+)。
/pdb	指定 .pdb 文件的文件名和位置。
/platform	限定此代码可以在其上运行的平台：x86、Itanium、x6 anycpu。
/preferreduilang	指定要用于编译器输出的语言。
/recurse	根据通配符规范，包括当前目录及子目录下的所有文件
/reference	从指定的程序集文件引用元数据。
/resource	嵌入指定的资源。
/ruleset:<file>	指定可禁用特定诊断的规则集文件。
/subsystemversion	指定可执行文件可以使用的子系统的最低版本。
/target	使用下列四个选项之一指定输出文件的格式：/target:appcontainerexe、/target:exe、/target:libra
/unsafe	允许不安全代码。
/utf8output	以 UTF-8 编码格式输出编译器消息。
/warn	设置警告等级 (0-4)。
/warnaserror	将特定警告报告为错误。

/win32icon	对输出使用此图标。
/win32manifest	指定自定义 win32 清单文件。
/win32res	指定 win32 资源文件 (.res)。

请参阅

C# Compiler Options

C# Compiler Options Listed by Category

How to: Set Environment Variables for the Visual Studio Command Line

<compiler> 元素

@ (C# Compiler Options)

@ 选项使您可以指定包含编译器选项和要编译的源代码文件的文件。

语法

```
@response_file
```

参数

response_file

一个列出编译器选项或要编译的源代码文件的文件。

备注

编译器在处理编译器选项和源代码文件时将它们视为是在命令行上指定的。

若要在一次编译中指定多个响应文件，请指定多个响应文件选项。例如：

```
@file1.rsp @file2.rsp
```

在响应文件中，多个编译器选项和源代码文件可以出现在同一行中。单个编译器选项的指定必须出现在同一行中（不能跨行）。响应文件可以带有以 # 符号开始的注释。

从响应文件内部指定编译器选项就如同在命令行发出这些命令。有关更多信息，请参见从命令行生成。

编译器在遇到命令选项时会进行处理。因此，命令行参数可以重写先前在响应文件中列出的选项。反之，响应文件中的选项也将重写先前在命令行或其他响应文件中列出的选项。

C# 提供 csc.rsp 文件，该文件与 csc.exe 文件位于同一目录中。有关 csc.rsp 的更多信息，请参见 /noconfig。

不能在 Visual Studio 开发环境中设置此编译器选项，也不能以编程方式对其进行更改。

以下几行来自一个示例响应文件：

```
# build the first output file
/target:exe /out:MyExe.exe source1.cs source2.cs
```

请参阅

C# Compiler Options

/addmodule (C# Compiler Options)

此选项将一个使用 target:module 开关创建的模块添加到当前的编译中。

语法

```
/addmodule:file[;file2]
```

参数

file, *file2*

包含元数据的输出文件。该文件不能包含程序集清单。若要导入多个文件，请用逗号或分号分隔文件名。

备注

运行时，所有用 **/addmodule** 添加的模块必须与输出文件在同一目录中。也就是说，编译时可以在任何目录中指定模块，但是在运行时模块必须在应用程序目录中。如果模块在运行时不在应用程序目录中，您将遇到 TypeLoadException。

file 不能包含程序集。例如，如果输出文件用 /target:module 创建，则其元数据可以用 **/addmodule** 导入。

如果输出文件用 **/target** 选项而不是 **/target:module** 创建，则其元数据无法用 **/addmodule** 导入，但是可以用 /reference 导入。

此编译器选项在 Visual Studio 中不可用；项目不能引用模块。另外，不能以编程方式更改此编译器选项。

编译源文件 input.cs 并从 metad1.netmodule 和 metad2.netmodule 中添加元数据来产生 out.exe：

```
csc /addmodule:metad1.netmodule;metad2.netmodule /out:out.exe input
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

多文件程序集

如何：生成多文件程序集

/appconfig (C# Compiler Options)

/appconfig 编译器选项使 C# 应用程序能够在程序集绑定时向公共语言运行时 (CLR) 指明程序集应用程序配置 (app.config) 文件的位置。

语法

```
/appconfig:file
```

参数

file

必选。包含程序集绑定设置的应用程序配置文件。

备注

/appconfig 的一种用途是升级方案，在该方案中，程序集必须同时引用特定引用程序集的 .NET Framework 版本和 .NET Framework for Silverlight 版本。例如，以 Windows Presentation Foundation (WPF) 编写的 XAML 设计器可能必须同时引用设计器用户界面的 WPF 桌面和 Silverlight 附带的 WPF 子集。相同设计器程序集必须访问这两个程序集。默认情况下，单独引用会导致编译器错误，因为程序集绑定将这两个程序集视为等效。

/appconfig 编译器选项使您能够通过使用 **<supportPortability>** 标记指定禁用默认行为的 app.config 文件的位置，如下面的示例所示。

<supportPortability PKT="7cec85d7bea7798e" enable="false"/>

编译器将该文件的位置传递给 CLR 的程序集绑定逻辑。

注意
如果您要使用 Microsoft Build Engine (MSBuild) 生成应用程序，可通过向 .csproj 文件添加属性标记来设置 **/appconfig** 编译器选项。若要使用已经设置在项目中 app.config 文件，添加属性标记 **<UseAppConfigForCompiler>** 到 .csproj 文件并设置其值到 **true**。若要指定一个不同的 app.config 文件，添加属性标记 **<AppConfigForCompiler>**并设置其值到文件的所在位置。

下面的示例演示一个 app.config 文件，它使应用程序可以具有对任何 .NET Framework 程序集的 .NET Framework 实现和 .NET Framework for Silverlight 实现的引用，该程序集存在于两个实现中。 **/appconfig** 编译器选项指定此

app.config 文件的位置。

```
<configuration>
      <runtime>
      <assemblyBinding>
            <supportPortability PKT="7cec85d7bea7798e" enable="fals
            <supportPortability PKT="31bf3856ad364e35" enable="fals
      </assemblyBinding>
      </runtime>
</configuration>
```

请参阅

.NET Framework Assembly Unification Overview

<supportPortability> 元素

C# Compiler Options Listed Alphabetically

/baseaddress (C# Compiler Options)

/baseaddress 选项使您可以指定加载 DLL 的首选基址。有关为何和何时使用此选项的更多信息，请参见 提高应用开始时间 和 拉里 Osterman 的网络日志。

语法

```
/baseaddress:address
```

参数

address

DLL 的基址。可以将该地址指定为十进制、十六进制或八进制数。

备注

DLL 的默认基址由 .NET Framework 公共语言运行时设置。

请注意：该地址中低位的数将会被舍入。例如，如果指定 0x11110001，它将被舍入为 0x11110000。

若要完成 DLL 的签名过程，请结合使用 SN.EXE 和 -R 选项。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“生成”属性页。
3. 单击“高级”按钮。
4. 修改“**DLL** 基址”属性。

   若要以编程方式设置此编译器选项，请参见 BaseAddress。

请参阅

ProcessModule.BaseAddress

C# Compiler Options

如何：修改项目属性和配置设置

/bugreport (C# Compiler Options)

指定应将调试信息放在文件中，供以后分析使用。

语法

```
/bugreport:file
```

参数

file

要包含 bug 报告的文件的名称。

备注

/bugreport 选项指定应将下面的信息放在 *file* 中：

- 编译中所有源代码文件的副本。
- 编译中使用的编译器选项列表。
- 有关编译器、运行时和操作系统的版本信息。
- 引用的程序集和模块（保存为十六进制数），.NET Framework 和 SDK 附带的程序集除外。
- 编译器输出（如果有）。
- 将会提示给您的问题的说明。
- 关于考虑问题应如何解决（就此将会向您提示）的说明。

如果此选项与 **/errorreport:prompt** 或 **/errorreport:send** 一起使用，则文件中的信息将发送至 Microsoft Corporation。

由于所有源代码文件的副本都将放置在 *file* 中，因此可能需要在尽可能短的程序中重现所怀疑的代码缺陷。

此编译器选项在 Visual Studio 中不可用，且不能通过编程方式进行更改。

注意，生成文件的内容会公开源代码，可能导致意外的信息泄露。

请参阅

C# Compiler Options

/errorreport (C# Compiler Options)

如何：修改项目属性和配置设置

/checked (C# Compiler Options)

/checked 选项指定，不在 checked 或 unchecked 关键字的范围内、并且产生的值超出数据类型范围的整数算法语句是否将导致运行时异常。

语法

```
/checked[+ | -]
```

备注

checked 或 **unchecked** 关键字范围内的整数算法语句不受 **/checked** 选项的影响。

如果不在 **checked** 或 **unchecked** 关键字范围内的整数算法语句产生的值超出数据类型范围，并且编译中使用了 **/checked+ (/checked)**，则该语句将在运行时导致异常。如果编译中使用的是 **/checked-**，则该语句在运行时不会导致异常。

该选项的默认值为 **/checked-**.使用 **/checked-** 的一个方案是生成大型应用程序。有时自动化的工具用于生成该应用程序，而且这种工具可以将 **/checked** 自动设置为 +。可通过指定 **/checked-** 重写工具的全局默认值。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的"属性"页。有关更多信息，请参见 "项目设计器"->"生成"页 (C#)。
2. 单击"生成"属性页。
3. 单击"高级"按钮。
4. 修改"检查算法上溢/下溢"属性。

若要以编程方式访问此编译器选项，请参见 CheckForOverflowUnderflow。

使用以下命令编译 t2.cs。命令中 /checked 的使用指定，任何不在 **checked** or **unchecked** 关键字范围内以及导致数据类型以外值的结果的文件中的整数算法语句，都会在测试时间内引发异常。

```
csc t2.cs /checked
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

Introduction to the Project Designer

/codepage (C# Compiler Options)

此选项指定在编译期间，当所需的页不是系统当前的默认代码页时使用的代码页。

语法

```
/codepage:id
```

参数

id

编译中用于所有源代码文件的代码页的 ID。

备注

如果编译的一个或多个源代码文件没有被创建为使用计算机上的默认代码页，则可以使用 **/codepage** 选项指定应使用哪个代码页。 **/codepage** 适用于编译中的所有源代码文件。

如果源代码文件是用计算机中的同一有效代码页创建的，或者是用 UNICODE 或 UTF-8 创建的，则不需要使用 **/codepage**。

有关如何 GetCPInfo 查找系统上支持哪些代码页的信息。

此编译器选项在 Visual Studio 中不可用，且不能通过编程方式进行更改。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/debug (C# Compiler Options)

/debug 选项使编译器生成调试信息并将其放置在一个或多个输出文件中。

语法

```
/debug[+ | -]
/debug:{full | pdbonly}
```

参数

+ | -

指定 **+** 或只指定 **/debug** 将使编译器生成调试信息，并将这些信息放置在一个程序数据库（.pdb 文件）中。指定 **-**（在不指定 **/debug** 时有效）将导致不创建任何调试信息。

full | *pdbonly*

指定编译器生成的调试信息类型。full 参数在没有指定 **/debug:pdbonly** 时有效，它允许将调试器附加到正在运行的程序。指定 pdbonly 允许在调试器中启动程序时进行源代码调试，但仅在正在运行的程序附加到调试器时才显示汇编程序。

备注

使用该选项创建调试版本。如果未指定 **/debug**、**/debug+** 或 **/debug:full**，则无法调试程序的输出文件。

如果使用 **/debug:full**，请注意 **/debug:full** 可能会对 JIT 优化代码的速度和大小有一些影响，并且可能会稍微影响代码质量。我们建议在生成发布代码时使用 **/debug:pdbonly** 或不使用 PDB。

> 注意
>
> **/debug:pdbonly** 和 **/debug:full** 的一个区别是：使用 **/debug:full** 时，编译器会发出 DebuggableAttribute，用于通知 JIT 编译器有可用的调试信息。因此，如果使用 **/debug:full**，而代码中包含设置为 false 的 DebuggableAttribute，则会发生错误。

有关如何配置应用程序的调试性能的信息，请参见使映像更易于调试。

若要更改 .pdb 文件的位置，请参见 /pdb (C# Compiler Options)。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“生成”属性页。
3. 单击“高级”按钮。
4. 修改“调试信息”属性。

有关如何以编程方式设置此编译器选项的信息，请参见 DebugSymbols。

将调试信息放置在输出文件 app.pdb 中：

```
csc /debug /pdb:app.pdb test.cs
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/define (C# Compiler Options)

/define 选项将 *name* 定义为程序的所有源代码文件中的一个符号。

语法

```
/define:name[;name2]
```

参数

name, *name2*

要定义的一个或多个符号的名称。

备注

使用 **/define** 选项与使用 #define 预处理器指令具有相同的效果，不同点在于编译器选项对项目中的所有文件都有效。在源文件中的 #undef 指令移除定义之前，符号在源文件中始终保持已定义状态。使用 /define 选项时，一个文件中的 **#undef** 指令不会影响到项目中的其他源代码文件。

可以将用该选项创建的符号与 #if、#else、#elif 和 #endif 一起使用，以按条件编译源文件。

/d 是 **/define** 的缩写形式。

通过使用分号或逗号分隔符号名称，可以使用 **/define** 定义多个符号。例如：

```
/define:DEBUG;TUESDAY
```

C# 编译器自身没有定义可以在源代码中使用的符号或宏；所有符号定义必须是用户定义的。

> 注意
>
> 与 C++ 这样的语言相同，C# **#define** 不允许给符号赋值。例如，不能使用 **#define** 创建宏或定义常数。如果您需要定义常数，请使用 **enum** 变量。如果您希望创建 C++ 样式的宏，请考虑其他方式，例如泛型。由于宏有容易出错的坏名声，因此 C# 不允许使用宏，而是提供了其他更安全的选择。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 在“生成”选项卡上，键入要在“条件编译符号”框中定义的符号。例如，如果要使用下面的代码示例，只需在文本框中键入 **xx**。

有关如何以编程方式设置此编译器选项的信息，请参见 DefineConstants。

```
// preprocessor_define.cs
// compile with: /define:xx
// or uncomment the next line
// #define xx
using System;
public class Test
{
    public static void Main()
    {
        #if (xx)
            Console.WriteLine("xx defined");
        #else
            Console.WriteLine("xx not defined");
        #endif
    }
}
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/delaysign (C# Compiler Options)

此选项将使编译器在输出文件中保留空间，以便以后添加数字签名。

语法

```
/delaysign[ + | - ]
```

参数

+ | -

如果需要完全签名的程序集，则使用 **/delaysign-**。如果只想将公钥放在程序集中，则使用 **/delaysign+**。默认值为 **/delaysign-**。

备注

如果不与 /keyfile 或 /keycontainer 一起使用，**/delaysign** 选项将无效。

如果要求完全签名的程序集，编译器将对包含清单（程序集元数据）的文件进行散列处理，并用私钥对该散列数据进行签名。产生的数字签名存储在包含清单的文件中。当果程序集的签名延迟时，编译器将不会计算和存储签名，但会在文件中保留空间以便以后添加签名。

例如，使用 **/delaysign+** 将允许测试人员把程序集放入全局缓存中。测试完成后，可以通过使用程序集链接器实用工具将私钥放入程序集中对程序集进行完全签名。

有关更多信息，请参见创建和使用具有强名称的程序集和延迟为程序集签名。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 修改“仅延迟签名”属性。

有关如何以编程方式设置此编译器选项的信息，请参见 DelaySign。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/doc (C# Compiler Options)

/doc 选项允许在 XML 文件中放置文档注释。

语法

```
/doc:file
```

参数

file

XML 的输出文件，由编译的源代码文件中的注释填充。

备注

在源代码文件中，可处理以下内容之前的文档注释，并将其添加到 XML 文件中：

- 用户定义的类型，如 class、delegate 或 interface
- 成员，如字段、事件、属性或方法

包含 Main 的源代码文件首先输出到 XML。

若要将生成的 .xml 文件用于Intellisense功能，请使该 .xml 文件的文件名与要支持的程序集同名，然后确保该 .xml 文件放入与该程序集相同的目录中。这样，在 Visual Studio 项目中引用程序集时，也可找到该 .xml 文件。有关更多信息，请参见提供代码注释。

除非用 /target:module 进行编译，否则 *file* 将包含 <assembly></assembly>，以指定包含编译输出文件的程序集清单的文件名。

> 注意
>
> /doc 选项适用于所有输入文件；或者，如果此选项是在“项目设置”中设置的，则适用于项目中的所有文件。若要禁用与特定的文件或代码段的文档注释相关的警告，请使用 #pragma warning。

有关从代码中的注释生成文档的方法，请参见建议的文档注释标记。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“生成”选项卡。
3. 修改“**XML** 文档文件”属性。

有关如何以编程方式设置此编译器选项的信息，请参见 DocumentationFile。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/errorreport (C# Compiler Options)

使用此选项可以方便地向 Microsoft 报告 C# 内部编译器错误。

> 注意
>
> 在 Windows Vista 和 Windows Server 2008 上，为 Visual Studio 做出的错误报告设置不会替代通过 Windows 错误报告 (WER) 做出的设置。WER 设置始终优先于 Visual Studio 的错误报告设置。

语法

```
/errorreport:{ none | prompt | queue | send }
```

参数

none

不收集有关内部编译器错误的报告，或不向 Microsoft 发送报告。

prompt

当您收到内部编译器错误时，提示您发送报告。在开发环境中编译应用程序时，**prompt** 是默认值。

queue

对错误报告进行排队。当使用管理凭据登录时，可以报告自上次登录以来出现的任何故障。如果每三天出现一次以上故障，则将不提示发送故障报告。在命令行编译应用程序时，**queue** 是默认值。

send

自动向 Microsoft 发送内部编译器错误报告。若要启用此选项，必须先同意 Microsoft 数据收集策略。首次在计算机上指定 **/errorreport:send** 时，编译器消息将引导您访问包含 Microsoft 数据收集策略的网站。

此选项取决于注册表设置。有关在注册表中如何设置相应值的信息，请参见 MSDN 网站上 How to Turn on Automatic Error Reporting in Visual Studio 2008 Command-line Tools（如何在 Visual Studio 2008 的命令行工具中打开自动错误报告）。

备注

当编译器无法处理源代码文件时，将导致内部编译器错误 (ICE)。当发生 ICE 时，编译器不生成输出文件或可用来修复代码的任何有用的诊断。

在以前的版本中，当收到 ICE 时，最好联系 Microsoft 产品支持服务以报告问题。现在通过使用 **/errorreport**，您可以向 Visual C# 团队提供 ICE 信息。错误报告有助于改进将来的编译器版本。

用户发送报告的能力取决于计算机和用户策略权限。

调试器有关错误的更多信息，参见Dr. Watson 的说明 (Drwtsn32.exe) 的工具窗口。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。有关详细信息，请参阅“项目设计器”->“生成”页 (C#)。
2. 单击“生成”属性页。
3. 单击“高级”按钮。
4. 修改“内部编译器错误报告”属性。

有关如何以编程方式设置此编译器选项的信息，请参阅 ErrorReport。

请参阅

C# Compiler Options

/filealign (C# Compiler Options)

/filealign 选项使您可以指定输出文件中的节大小。

语法

```
/filealign:number
```

参数

number

指定输出文件中节大小的值。有效值为 512、1024、2048、4096 和 8192。这些值以字节为单位。

备注

每个节将在是 **/filealign** 值的倍数的边界上对齐。没有固定的默认值。如果未指定 **/filealign**，则公共语言运行时在编译时将选取一个默认值。

通过指定节的大小，可以影响输出文件的大小。修改节的大小可能对将在较小设备上运行的程序有用。

使用 DUMPBIN 查看有关输出文件中的节信息。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“生成”属性页。
3. 单击“高级”按钮。
4. 修改“文件对齐”属性。

有关如何以编程方式设置此编译器选项的信息，请参见 FileAlignment。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/fullpaths (C# Compiler Options)

/fullpaths 选项可使编译器在列出编译错误和警告时指定文件的完整路径。

语法

```
/fullpaths
```

备注

默认情况下，编译所产生的错误和警告指定包含错误的文件名。 **/fullpaths** 选项使编译器指定文件的完整路径。

此编译器选项在 Visual Studio 中不可用，且不能通过编程方式进行更改。

请参阅

C# Compiler Options

/help, /? (C# Compiler Options)

该选项将编译器选项的列表和每个选项的简短说明发送到 stdout。

语法

```
/help
/?
```

备注

如果编译中包含该选项，将不会创建输出文件，也不会进行编译。

此编译器选项在 Visual Studio 中不可用，且不能通过编程方式进行更改。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/highentropyva (C# Compiler Options)

/highentropyva 编译器选项告诉 Windows 内核：特定的可执行文件是否支持高熵地址空间格式随机化 (ASLR)。

语法

```
/highentropyva[+ | -]
```

参数

+ | -

此选项指定一位 64 可执行或由 /platform: 了 编译器选项指示可执行的支持高熵虚拟地址空间。默认情况下，该选项是关闭的。使用 **/highentropyva+** 或 **/highentropyva** 中启用它。

备注

当随机化进程的地址空间作为格式 ASLR 一部分时，**/highentropyva** 选项允许 Windows 内核的兼容版本使用高度熵。使用高熵意味着地址的大部分分配到内存区域 (栈和堆）。因此，猜想特定的内存区域中的位置更加困难。

当指定 **/highentropyva** 编译器选项时，可执行的目标和它所依赖的任何模块在测试运行作为 64 位进程时，必须能够处理大于 4 GB 的指针值 (GB)，。

有关ASLR的更多信息，请参见 Mitigating Software Vulnerabilities。

/keycontainer (C# Compiler Options)

指定加密密钥容器的名称。

语法

```
/keycontainer:string
```

参数

string

强名称密钥容器的名称。

备注

使用 **/keycontainer** 选项时，通过将来自所指定容器的公钥插入到程序集清单中并且用私钥签名最终程序集，编译器可创建可共享的组件。生成密钥文件，请在命令行上键入sn -k *file*。sn -i 安装密钥对为容器。

如果使用 /target:module 进行编译，则将密钥文件的名称保存在模块中，并在使用 /addmodule 编译程序集时将其包含到该程序集中。

还可以将此选项指定为任何 Microsoft 中间语言 (MSIL) 模块的源代码中的自定义特性 (System.Reflection.AssemblyKeyNameAttribute)。

也可以通过 /keyfile 将加密信息传递给编译器。如果希望将公钥添加到程序集清单中，但将程序集的签名延迟到该程序集通过测试，则请使用 /delaysign。

有关更多信息，请参见创建和使用具有强名称的程序集和延迟为程序集签名。

在 Visual Studio 开发环境中设置此编译器选项

1. 此编译器选项在 Visual Studio 开发环境中不可用。

可以使用 AssemblyKeyContainerName 以编程方式访问此编译器选项。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/keyfile (C# Compiler Options)

指定包含加密密钥的文件名。

语法

```
/keyfile:file
```

参数

术语	定义
file	包含强名称密钥的文件的名称。

备注

使用此选项时，编译器将从指定的文件将公钥插入程序集清单中，然后使用私钥对最终程序集进行签名。若要生成密钥文件，请在命令行上键入 sn -k *file*。

如果使用 **/target:module** 进行编译，则将密钥文件的名称保存在模块中，并在使用 /addmodule 编译程序集时将其包含到创建的程序集中。

也可以使用 /keycontainer 将加密信息传递给编译器。如果需要部分签名的程序集，则使用 /delaysign。

如果在相同的编译中指定了 /keyfile 和 /keycontainer（通过命令行选项或自定义特性），编译器将首先尝试密钥容器。如果成功，则使用密钥容器中的信息对程序集进行签名。如果编译器没有找到密钥容器，则将尝试用 /keyfile 指定的文件。如果成功，则使用密钥文件中的信息对程序集进行签名，并且将把密钥信息安装到密钥容器中（类似 sn -i），这样，下次编译时密钥容器将是有效的。

请注意，密钥文件可能只包含公钥。

有关更多信息，请参见创建和使用具有强名称的程序集和延迟为程序集签名。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“签名”属性页。
3. 修改“选择强名称密钥文件”属性。

可以使用 AssemblyOriginatorKeyFile 以编程方式访问此编译器选项。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/langversion (C# Compiler Options)

导致编译器只接受所选 C# 语言规范中包含的语法。

语法

```
/langversion:option
```

参数

option

以下为有效值：

选项	含义
default	编译器接受所有有效的语言语法。
ISO-1	编译器只接受 ISO/IEC 23270:2003 C# 语言规范中包含的语法。
ISO-2	编译器只接受 ISO/IEC 23270:2006 C# 语言规范中包含的语法。此规范上在ISO网站。
3	编译器只接受 3.0 版本 C# 语言规范中包含的语法。

备注

C# 应用程序所引用的元数据不受 **/langversion** 编译器选项的影响。

由于 C# 编译器的每个版本都包含语言规范的扩展，因此 **/langversion** 不提供该编译器早期版本的等效功能。

无论您使用的是何种 **/langversion** 设置，都将使用当前版本的公共语言运行时来创建 .exe 或 .dll。这种情况的一个例外是友元程序集和 /moduleassemblyname (C# Compiler Option)，它们用于 **/langversion:ISO-1**。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“生成”属性页。
3. 单击“高级”按钮。

4. 修改“语言版本”属性。

有关如何以编程方式设置此编译器选项的信息，请参阅 LanguageVersion。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

C# 语言规范

/lib (C# Compiler Options)

/lib 选项指定通过 /reference (C# Compiler Options) 选项引用的程序集的位置。

语法

```
/lib:dir1[,dir2]
```

参数

dir1

在当前工作目录（调用编译器的目录）或公共语言运行时的系统目录中未找到引用的程序集时，编译器将在其中进行查找的目录。

dir2

要在其中搜索程序集引用的一个或多个附加目录。用逗号分隔每个附加目录的名称，中间不要有空格。

备注

编译器按以下顺序搜索未完全限定的程序集引用：

1. 当前工作目录。该目录为从其调用编译器的目录。
2. 公共语言运行时系统目录。
3. 由 /lib 指定的目录。
4. 由 LIB 环境变量指定的目录。

使用 /reference 指定程序集引用。

/lib 是累加的；每一次指定的值都追加到以前的值中。

另一种使用 /lib 的方法是，将任何所需的程序集复制到工作目录中；这使您可以仅将程序集名称传递给 /reference。然后可以从工作目录中删除这些程序集。由于程序集清单中未指定依赖程序集的路径，因此应用程序可以在目标计算机上启动，然后查找并使用全局程序集缓存中的程序集。

编译器可以引用程序集并不表示公共语言运行时可以在运行时找到并加载程序集。有关运行时如何搜索引用的程序集的详细信息，请参见 运行时如何定位程序集。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性页”对话框。
2. 单击“引用路径”属性页。
3. 修改列表框的内容。

有关如何以编程方式设置此编译器选项的信息，请参见 ReferencePath。

编译 t2.cs 以创建 .exe 文件。编译器将在工作目录和驱动器 C 上根目录中查找程序集引用。

```
csc /lib:c:\ /reference:t2.dll t2.cs
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/link (C# Compiler Options)

使编译器将指定程序集中的 COM 类型信息提供给当前正在编译的项目使用。

语法

```
/link:fileList
// -or-
/l:fileList
```

参数

fileList

必选。程序集文件名的逗号分隔列表。如果文件名包含空格，则将该文件名置于引号中。

备注

通过 **/link** 选项，可以部署包含嵌入类型信息的应用程序。这样，该应用程序无需引用运行时程序集，就可以使用运行时程序集中实现嵌入类型信息的类型。如果发布了各种版本的运行时程序集，则包含嵌入类型信息的应用程序可以使用各种版本，而不必重新编译。有关示例，请参见演练：嵌入托管程序集中的类型（C# 和 Visual Basic）。

当使用 COM 互操作时，使用 **/link** 选项尤为有用。通过嵌入 COM 类型，您的应用程序不再要求目标计算机上存在主互操作程序集 (PIA)。 **/link** 选项指示编译器将所引用的互操作程序集中的 COM 类型信息嵌入到所生成的编译代码中。该 COM 类型由 CLSID (GUID) 值标识。因此，您的应用程序可以在安装有 CLSID 值相同的相同 COM 类型的目标计算机上运行。自动执行 Microsoft Office 的应用程序就是一个很好的示例。由于诸如 Office 之类的应用程序通常在不同的版本中保留相同的 CLSID 值，因此只要目标计算机上安装有 .NET Framework 4 或更高版本，的应用程序就可以使用所引用的 COM 类型，并使用包含在所引用 COM 类型中的方法、属性或事件。

/link 选项仅嵌入接口、结构和委托。不支持嵌入 COM 类。

> 注意
>
> 当在代码中创建嵌入 COM 类型的实例时，必须使用适当的接口创建该实例。如果尝试使用 CoClass 创建嵌入 COM 类型的实例，则会导致错误。

若要在 Visual Studio 中设置 **/link** 选项，请添加程序集引用并将 **Embed Interop Types** 属性设置为 **true**。 **Embed Interop Types** 属性的默认值为 **false**。

如果链接至一个 COM 程序集（程序集 A），而其本身又引用另一个 COM 程序集（程序集 B），则在符合下列任一条件的情况下也必须链接至程序集 B：

- 程序集 A 中的类型继承自程序集 B 中的类型或实现程序集 B 中的接口。
- 调用具有程序集 B 中的返回类型或参数类型的字段、属性、事件或方法。

与 /reference 编译器选项类似，**/link** 编译器选项使用 Csc.rsp 响应文件，该文件引用经常用到的 .NET Framework 程序集。如果希望编译器不要使用 Csc.rsp 文件，请使用 /noconfig 编译器选项。

/link 的缩写形式是 **/l**。

泛型和嵌入类型

以下各节介绍在嵌入互操作类型的应用程序中使用泛型类型时所存在的限制。

泛型接口

不能使用从互操作程序集中嵌入的泛型接口。这将在下面的示例中显示。

```
// The following code causes an error if ISampleInterface is an emb
ISampleInterface<SampleType> sample;
```

具有泛型参数的类型

对于具有泛型参数并且该参数类型是从互操作程序集嵌入的类型，如果该类型来自外部程序集，则不能使用该类型。此限制不适用于接口。例如，请考虑在 Microsoft.Office.Interop.Excel 程序集中定义的 Range 接口。如果库从 Microsoft.Office.Interop.Excel 程序集嵌入互操作类型并公开一个返回泛型类型的方法，该泛型类型的参数类型为 Range 接口，则该方法必须返回一个泛型接口，如下面的代码示例中所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.Excel;

public class Utility
{
    // The following code causes an error when called by a client a
    public List<Range> GetRange1() {
```

```
}

// The following code is valid for calls from a client assembly.
public IList<Range> GetRange2() {
```

```
    }
}
```

在下面的示例中，客户端代码可调用在不出错的情况下返回 IList 泛型接口的方法。

```
public class Client
{
    public void Main()
    {
        Utility util = new Utility();

        // The following code causes an error.
        List<Range> rangeList1 = util.GetRange1();

        // The following code is valid.
        List<Range> rangeList2 = (List<Range>)util.GetRange2();
    }
}
```

下面的代码编译源文件 OfficeApp.cs 并引用 COMData1.dll 和 COMData2.dll 中的程序集来生成 OfficeApp.exe。

```
csc /link:COMData1.dll,COMData2.dll /out:OfficeApp.exe OfficeApp.cs
```

请参阅

C# Compiler Options

演练：嵌入托管程序集中的类型（C# 和 Visual Basic）

/reference (C# Compiler Options)

/noconfig (C# Compiler Options)

Command-line Building With csc.exe

互操作性概述（C# 编程指南）

/linkresource (C# Compiler Options)

在输出文件中创建指向 .NET Framework 资源的链接。该资源文件未添加到输出文件中。这与 /resource 选项不同，后者会将资源文件嵌入到输出文件中。

语法

```
/linkresource:filename[,identifier[,accessibility-modifier]]
```

参数

filename

要从程序集链接的 .NET Framework 资源文件。

identifier（可选）

资源的逻辑名称；用于加载资源的名称。默认为文件的名称。

accessibility-modifier（可选）

资源的可访问性：公共或私有。默认为公共。

备注

默认情况下，如果链接的资源是用 C# 编译器创建的，则链接资源在程序集中就是公共的。若要使这些资源成为私有的，请将 **private** 指定为可访问性修饰符。不允许使用 **public** 或 **private** 以外的其他修饰符。

/linkresource 需要除 **/target:module** 选项之外的 /target 选项之一。

如果 *filename* 是由例如 Resgen.exe 创建或在开发环境中创建的 .NET Framework 资源文件，则可以通过 System.Resources 命名空间中的成员访问该文件。有关更多信息，请参见 System.Resources.ResourceManager。对于所有其他资源，请使用 Assembly 类中的 **GetManifestResource*** 方法在运行时访问资源。

在 *filename* 中指定的文件可以为任何格式。例如，您可能想将本机 DLL 设置为程序集的一部分，以便可将其安装到全局程序集缓存中，并且可从程序集中的托管代码访问它。下面的第二个示例演示如何执行此操作。您可以在程序集链接器中执行相同的操作。下面的第三个示例演示如何执行此操作。有关更多信息，请参见 Al.exe（程序集链接器）和 使用程序集和全局程序集缓存。

/linkres 是 **/linkresource** 的缩写形式。

此编译器选项在 Visual Studio 中不可用，且不能通过编程方式进行更改。

编译 in.cs 并链接到资源文件 rf.resource：

```
csc /linkresource:rf.resource in.cs
```

将 A.cs 编译为 DLL，链接到本机 DLL N.dll，并将输出放置在全局程序集缓存 (GAC) 中。在此示例中，A.dll 和 N.dll 都驻留在 GAC 中。

```
csc /linkresource:N.dll /t:library A.cs
gacutil -i A.dll
```

此示例和前一个示例执行的是同样的操作，但使用的是程序集链接器选项。

```
csc /t:module A.cs
al /out:A.dll A.netmodule /link:N.dll
gacutil -i A.dll
```

请参阅

C# Compiler Options

Al.exe（程序集链接器）

使用程序集和全局程序集缓存

如何：修改项目属性和配置设置

/main (C# Compiler Options)

如果有多个类包含 **Main** 方法，此选项指定包含程序入口点的类。

语法

```
/main:class
```

参数

class

包含 **Main** 方法的类型。

备注

如果编译包括多个具有 Main 方法的类型，可以指定哪个类型包含要用作程序入口点的 **Main** 方法。

此选项用在编译 .exe 文件时。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“应用程序”属性页。
3. 修改“启动对象”属性。

    若要以编程方式设置此编译器选项，请参见 StartupObject。

编译 t2.cs 和 t3.cs，并指定将在 Test2 中找到 **Main** 方法：

```
csc t2.cs t3.cs /main:Test2
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/moduleassemblyname (C# Compiler Option)

指定一个程序集，其非公共类型可由 netmodule 访问。

语法

```
/moduleassemblyname:assembly_name
```

参数

assembly_name

程序集的名称，.netmodule 可以访问该程序集的非公共类型。

备注

应使用**/moduleassemblyname**，当生成 .netmodule，且满足以下条件的位置：

- netmodule 需要具有访问现有程序集中非公共类型的权限。
- 知道 .netmodule 将生成的程序集的名称。
- 现存的程序集已经被授予.netmodule 将生成到的程序集友元程序集访问权限。

有关建立.netmodule 的更多信息，请参见 /target:module (C# Compiler Options)。

有关友元程序集的更多信息，请参见 友元程序集（C# 和 Visual Basic）。

此选项在开发环境中不可用；它仅在从命令行编译时才可用。

此编译器选项在 Visual Studio 中不可用，且不能通过编程方式进行更改。

此示例生成具有私有类型的程序集，并且该程序集授予称为 csman_an_assembly 的程序集友元程序集访问权限。

```
// moduleassemblyname_1.cs
// compile with: /target:library
using System;
using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo ("csman_an_assembly")]

class An_Internal_Class
{
    public void Test()
    {
        Console.WriteLine("An_Internal_Class.Test called");
    }
}
```

该例建立一个可访问程序集 moduleassemblyname_1.dll 中非公共类型的.netmodule。通过了解此 .netmodule 将生成到调用 csman_an_assembly的程序集，可以指定 **/moduleassemblyname**，允许 .netmodule 在已经别授予 csman_an_assembly的友元程序集访问权限的程序集中访问非公共类型。

```
// moduleassemblyname_2.cs
// compile with: /moduleassemblyname:csman_an_assembly /target:modu
class B {
    public void Test() {
        An_Internal_Class x = new An_Internal_Class();
        x.Test();
    }
}
```

此代码示例通过引用以前生成的程序集和 .netmodule， 生成程序集 csman_an_assembly。

```
// csman_an_assembly.cs
// compile with: /addmodule:moduleassemblyname_2.netmodule /referen
class A {
    public static void Main() {
        B bb = new B();
        bb.Test();
    }
}
```

```
已调用 An_Internal_Class.Test
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/noconfig (C# Compiler Options)

/noconfig 选项通知编译器不要使用 csc.rsp 文件进行编译，该文件与 csc.exe 文件位于同一目录中，并从该目录中加载。

语法

```
/noconfig
```

备注

csc.rsp 文件引用 .NET Framework 附带的所有程序集。Visual Studio .NET 开发环境包括的实际引用具体取决于项目类型。

可以修改 csc.rsp 文件并指定其他编译器选项，而这些选项是那些应该包括在使用 csc.exe 的来自命令行的每次编译中的选项（**/noconfig** 选项除外）。

编译器会保留上次传递给 **csc** 命令的选项。因此，命令行上的任何选项都会重写 csc.rsp 文件中同一选项的设置。

如果不希望编译器查找并使用 csc.rsp 文件中的设置，请指定 **/noconfig**。

此编译器选项在 Visual Studio 中不可用，且不能通过编程方式进行更改。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/nologo (C# Compiler Options)

/nologo 选项在编译器启动时取消显示登录版权标志，并在编译期间取消显示信息性消息。

语法

```
/nologo
```

备注

此选项在开发环境中不可用；它仅在从命令行编译时才可用。

此编译器选项在 Visual Studio 中不可用，且不能通过编程方式进行更改。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/nostdlib (C# Compiler Options)

/nostdlib 禁止导入定义整个 System 命名空间的 mscorlib.dll。

语法

```
/nostdlib[+ | -]
```

备注

如果您希望定义或创建自己的 System 命名空间和对象，请使用该选项。

如果不指定 **/nostdlib**，mscorlib.dll 将导入程序（与指定 **/nostdlib-** 相同）。指定 **/nostdlib** 与指定 **/nostdlib+** 的作用相同。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“生成”属性页。
3. 单击“高级”按钮。
4. 修改“不引用 **mscorlib.dll**”属性。

有关如何以编程方式设置此编译器选项的信息，请参见 NoStdLib。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/nowarn (C# Compiler Options)

/nowarn 选项允许您禁止编译器显示一个或多个警告。用逗号分隔多个警告编号。

语法

```
/nowarn:number1[,number2,...]
```

参数

number1, *number2*

您希望编译器取消的警告编号。

备注

只应指定警告标识符的数值部分。例如，若要取消 CS0028，可以指定 /nowarn:28。

编译器在无人参与的模式下忽略传递给 **/nowarn** 的警告编号，这些警告编号在先前版本中有效但已从编译器中移除。例如，CS0679 在 Visual Studio .NET 2002 中有效但后来被删除。

/nowarn 选项无法禁止显示以下警告：

- 编译器警告（等级 1）CS2002
- 编译器警告（等级 1）CS2023
- 编译器警告（等级 1）CS2029

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。有关详细信息，请参见“项目设计器”->“生成”页 (C#)。
2. 单击“生成”属性页。
3. 修改“取消警告”属性。

有关如何以编程方式设置此编译器选项的信息，请参阅 DelaySign。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

C# Compiler Errors

/nowin32manifest (C# Compiler Options)

使用 **/nowin32manifest** 选项可指示编译器不将任何应用程序清单嵌入到可执行文件中。

语法

```
/nowin32manifest
```

备注

使用此选项时，除非在 Win32 资源文件或以后的生成步骤中提供应用程序清单，否则应用程序会受到 Windows Vista 上虚拟化的影响。有关虚拟化的更多信息，请参见。80efa4c7-3904-45c5-82e8-2d558fe67db9

在 Visual Studio 的“应用程序属性”页中，通过在“清单”下拉列表中选择“创建不带清单的应用程序”选项来设置此选项。有关更多信息，请参见“项目设计器”->“应用程序”页 (C#)。

有关创建清单的更多信息，请参见 /win32manifest (C# Compiler Options)。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/optimize (C# Compiler Options)

/optimize 选项启用或禁用由编译器执行以使输出文件更小、更快和更有效的优化。

语法

```
/optimize[+ | -]
```

备注

/optimize 选项还通知公共语言运行时在运行时优化代码。

默认情况下，将禁用优化。指定 **/optimize+** 以启用优化。

在生成要由程序集使用的模块时，请使用与该程序集的设置相同的 **/optimize** 设置。

/o 是 **/optimize** 的缩写形式。

可以组合 **/optimize** 和 /debug 选项。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“生成”属性页。
3. 修改“优化代码”属性。

有关如何以编程方式设置此编译器选项的信息，请参见 Optimize。

生成 t2.cs并允许编译器优化：

```
csc t2.cs /optimize
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/out (C# Compiler Options)

/out 选项指定输出文件的名称。

语法

```
/out:filename
```

参数

filename

由编译器创建的输出文件的名称。

备注

在命令行中，可以为编译指定多个输出文件。编译器希望在 **/out** 选项之后找到一个或多个源代码文件。这样一来，所有的源代码文件都将被编译到 **/out** 选项所指定的那个输出文件中。

指定要创建的文件的完整名称和扩展名。

如果不指定输出文件的名称：

- .exe 文件将从包含 **Main** 方法的源代码文件中获取其名称。
- 一个.dll或者netmodule 将从第一个源代码文件中获取其名称。

用于编译一个输出文件的源代码文件不能在相同的编译中用作另一个输出文件的编译。

当在命令行编译中产生多个输出文件时，请记住其中只有一个输出文件可以是程序集并且只有用 **/out** 隐式或显式指定的第一个输出文件才是程序集。

作为编译的一部分产生的任何模块都变成与编译中产生的所有程序集相关的文件。使用 ildasm.exe 浏览程序集清单以查看相关的文件。

为使 EXE 成为友元程序集的目标，需要 /out 编译器选项。有关更多信息，请参见友元程序集（C# 和 Visual Basic）。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。

2. 单击“应用程序”属性页。
3. 修改“程序集名称”属性。

   若要以编程方式设置此编译器选项：OutputFileName 是只读属性，它由项目类型（exe、库等）和程序集名称的组合确定。必须修改这两个属性中的一个或全部才能设置输出文件名。

编译 t.cs 并创建输出文件 t.exe，同时又生成 t2.cs 并创建模块输出文件 mymodule.netmodule：

```
csc t.cs /out:mymodule.netmodule /target:module t2.cs
```

请参阅

C# Compiler Options

友元程序集（C# 和 Visual Basic）

如何：修改项目属性和配置设置

/pdb (C# Compiler Options)

/pdb 编译器选项指定调试符号文件的名称和位置。

语法

```
/pdb:filename
```

参数

filename

调试符号文件的名称和位置。

备注

当您指定 /debug (C# Compiler Options) 时，编译器将在创建输出文件（.exe 或 .dll）的同一目录下创建一个与输出文件同名的 .pdb 文件。

/pdb 允许您为 .pdb 文件指定非默认的文件名和位置。

不能在 Visual Studio 开发环境中设置此编译器选项，也不能以编程方式对其进行更改。

编译 t.cs 并创建一个名为 tt.pdb 的 .pdb 文件：

```
csc /debug /pdb:tt t.cs
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/platform (C# Compiler Options)

指定公共语言运行时 (CLR) 的哪个版本可以运行程序集。

语法

```
/platform:string
```

参数

string

anycpu(默认值)，anycpu32bitpreferred，ARM，x64，x86、或者Itanium。

备注

- **anycpu**（默认值）将编译程序集为使其在任意平台上都可以运行。在任何可能的时候，应用程序作为 64 位进程运行；仅当该模式只可用时，才会回退到 32 位。
- **anycpu32bitpreferred** 将程序集编译成可在任何平台上运行。应用程序在支持 64 位和 32 位应用程序的系统以32 位模式运行。可以仅仅为针对 .NET Framework 4.5 的项目指定此选项。
- **ARM**编译程序集，以便可以在具有高级 RISC (ARM) 计算机处理器的计算机运行。
- **x64** 将程序集编译成可由 64 位公共语言运行库在支持 AMD64 或 EM64T 指令集的计算机上运行。
- **x86**将程序集编译为由与 x86 兼容的 32 位公共语言运行时运行。
- **Itanium**将程序集编译为由采用 Itanium 处理器的计算机上的 64 位公共语言运行时运行。

在 64 位 Windows 操作系统上：

- 用 **/platform:x86** 编译的程序集将在运行于 WOW64 下的 32 位 CLR 上执行。
- 用 **/platform:anycpu** 编译的 DLL 将在加载该进程的同一 CLR 上执行。
- 用 **/platform:anycpu** 编译的可执行文件将在 64 位 CLR 上执行。

- 用 **/platform:anycpu32bitpreferred** 编译的可执行文件将在 32 位 CLR 上执行。

anycpu32bitpreferred 设置只对为可执行 (.exe) 文件有效，因此，它需要 .NET Framework 4.5。

有关开发在 Windows 64 位操作系统上运行的应用程序的更多信息，请参见 64 位应用程序。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“生成”属性页。
3. 修改 目标平台 属性，因此，若要面向 .NET Framework 4.5，选择或清除首选 **32** 位 复选框。

Note **/platform** 在 Visual C# 速成版开发环境中不可用。

有关如何以编程方式设置此编译器选项的信息，请参见 PlatformTarget。

下面的示例演示如何使用 **/platform**选项来指定应只由64 位 Windows 操作系统上的 64 位 CLR 运行的应用程序。

```
csc /platform:anycpu filename.cs
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/preferreduilang (C# Compiler Options)

使用 **/preferreduilang** 编译器选项，则可指定 C#编译器显示输出的语言，如错误消息。

语法

```
/preferreduilang: language
```

参数

language

编译器输出用到的语言的语言名称 。

备注

可以使用 **/preferreduilang** 编译器选项指定C# 编译器用于错误消息和其他命令行输出的语言。如果语言的语言包不安装，使用操作系统的语言设置，因此，不会报告错误。

```
csc.exe /preferreduilang:ja-JP
```

要求

请参阅

C# Compiler Options

/recurse (C# Compiler Options)

/recurse 选项使您可以编译指定目录 (dir) 或项目目录的所有子目录中的源代码文件。

语法

```
/recurse:[dir\]file
```

参数

dir（可选）

搜索开始的目录。如果未指定此目录，则搜索从项目目录开始。

file

要搜索的文件。允许使用通配符字符。

备注

/recurse 选项使您可以编译指定目录 (*dir*) 或项目目录的所有子目录中的源代码文件。

可以在文件名中使用通配符来编译项目目录中所有匹配的文件，而不需使用 **/recurse**。

此编译器选项在 Visual Studio 中不可用，且不能通过编程方式进行更改。

编译当前目录中的所有 C# 文件：

```
csc *.cs
```

编译 dir1\dir2 目录及其任何子目录中的所有 C# 文件，并生成 dir2.dll：

```
csc /target:library /out:dir2.dll /recurse:dir1\dir2\*.cs
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/reference (C# Compiler Options)

/reference 选项导致编译器将指定文件中的 公开 类型信息导入到当前项目中，从而使您可以从指定的程序集文件引用元数据。

语法

```
/reference:[alias=]filename
/reference:filename
```

参数

filename

包含程序集清单的文件的名称。若要导入多个文件，请为每个文件包括一个单独的 **/reference** 选项。

alias

一个有效的 C# 标识符，表示将包含程序集中所有命名空间的根命名空间。

备注

若要从多个文件导入，请为每个文件包括一个 **/reference** 选项。

导入的文件必须包含一个清单；输出文件必须已使用 /目标:模块 以外的 /目标 选项之一编译过。

/r 是 **/reference** 的缩写形式。

使用 /addmodule 从不包含程序集清单的输出文件导入元数据。

如果您引用了一个程序集（程序集 A），其本身又引用了另一个程序集（程序集 B），则在下列情况下需要引用程序集 B：

- 使用来自程序集 A 的类型继承自程序集 B 中的类型或实现程序集 B 中的接口。
- 调用具有程序集 B 中的返回类型或参数类型的字段、属性、事件或方法。

使用 /lib 指定一个或多个程序集引用所在的目录。 **/lib** 主题还讨论了编译器在哪些目录中搜索程序集。

为使编译器可以识别程序集（而不是模块）中的某个类型，需要强制解析此类型，这可以通过定义此类型的实例来完成。还有其他方法可为编译器解析程序集中的类型名称：例如，如果您从程序集中的类型继承，编译器就能识别类型名称。

有时必须从一个程序集内部引用同一组件的两个不同版本。为此，请在每个文件的 **/reference** 开关上使用 alias 子选项，以区分两个文件。此别名将用作组件名称的限定符，并解析为其中一个文件中的组件。

默认情况下使用 csc 响应 (.rsp) 文件，该文件引用常用的 .NET Framework 程序集。如果希望编译器不要使用 csc.rsp，请使用 /noconfig。

> 注意
>
> 在 Visual Studio 中，请使用“添加引用”对话框。有关更多信息，请参见如何：使用“添加引用”对话框添加或移除引用。在 Visual Studio 2010 和更高版本中，若要确保通过使用 **/reference** 添加引用与通过使用“添加引用”对话框添加引用之间的行为等效，必须将您要添加的程序集的“嵌入互操作类型”属性设置为**“False”**。**“True”**为该属性的默认值。

示例

本示例演示如何使用外部别名功能。

编译源文件，并从先前已编译的 grid.dll 和 grid20.dll 中导入元数据。这两个 DLL 包含同一组件的不同版本，您将使用两个带有别名选项的 **/reference** 编译源文件。这两个选项如下所示：

/reference:GridV1=grid.dll 和 /reference:GridV2=grid20.dll

这将设置外部别名“GridV1”和“GridV2”，您将在程序中通过外部语句使用它们：

```
extern alias GridV1;
extern alias GridV2;
// Using statements go here.
```

完成此操作后，您可以通过在控件名称前添加 GridV1 前缀来引用 grid.dll 中的网格控件，如下所示：

```
GridV1::Grid
```

此外，您可以通过在控件名称前添加 GridV2 前缀来引用 grid20.dll 中的网格控件，如下所示：

```
GridV2::Grid
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/resource (C# Compiler Options)

将指定的资源嵌入到输出文件中。

语法

```
/resource:filename[,identifier[,accessibility-modifier]]
```

参数

filename

要在输出文件中嵌入的 .NET Framework 资源文件。

identifier（可选）

资源的逻辑名称；用于加载资源的名称。默认为文件的名称。

accessibility-modifier（可选）

资源的可访问性：公共或私有。默认为公共。

备注

使用 /linkresource 将资源链接到程序集，但不将资源文件添加到输出文件中。

默认情况下，如果资源是用 C# 编译器创建的，则该资源在程序集中就是公共的。若要使这些资源成为私有的，请将 **private** 指定为可访问性修饰符。不允许使用 **public** 或 **private** 以外的可访问性。

如果 *filename* 是由例如 Resgen.exe 创建或在开发环境中创建的 .NET Framework 资源文件，则可以通过 System.Resources 命名空间中的成员访问该文件。有关更多信息，请参见 System.Resources.ResourceManager。对于所有其他资源，请使用 Assembly 类中的 **GetManifestResource*** 方法在运行时访问资源。

/res 是 **/resource** 的缩写形式。

资源在输出文件中的顺序是由命令行上指定的顺序决定的。

在 Visual Studio 开发环境中设置此编译器选项

1. 将资源文件添加到项目中。

2. 在“解决方案资源管理器”中，选择要嵌入的文件。
3. 在“属性”窗口中，为该文件选择“生成操作”。
4. 将“生成操作”设置为“嵌入的资源”。

有关如何以编程方式设置此编译器选项的信息，请参见 BuildAction。

编译 in.cs 并附加资源文件 rf.resource：

```
csc /resource:rf.resource in.cs
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/subsystemversion (C# Compiler Options)

指定产生的可执行文件可以运行的子系统的最低版本，从而确定执行文件可以运行的Windows的版本。通常，此选项确保可执行文件可以利用 Windows早期版本中不可以使用的特定安全功能。

注意
若要指定的子系统，请使用 /target 编译器选项。

语法

```
/subsystemversion:major.minor
```

参数

major.minor

子系统的最低要求的版本，主要和次要版本的点表示形式表示。例如，可以指定一个不可以运行在 Windows 7以前的操作系统的应用程序，如本主题后面的表所述，如果将此选项值设置为 6.01。必须为 *major* 和 *minor* 指定值为整数。

前导零的 *minor* 版本不更改版本，但是填充的零更改。例如，6.1 和 6.01 都引用同一个版本，但是，6.10 引用不同版本的。建议把次版本表示为两个数字以避免混淆。

备注

下表列出常用 Windows 子系统版本。

Windows 版本	子系统版本
Windows 2000	5.00
Windows XP	5.01
Windows Server 2003	5.02
Windows Vista	6.00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02

默认值

/subsystemversion 编译器选项的默认值取决于列表下面的条件：

- 如果以下列表中的任何编译器选项被设置，默认值为 6.02:
  - /target:appcontainerexe
  - /target:winmdobj
  - /platform:arm
- 如果使用 MSBuild，默认值为 6.00，则以.NET Framework 4.5为目的，并且未设置此列表之前指定的任何编译器选项。
- 如果以上条件都不符合，默认值为 4.00。

设置此选项

若要在 Visual Studio 中设置 **/subsystemversion** 编译器选项，则必须在MSBuild XML中打开 .csproj 文件和指定 **SubsystemVersion** 属性值。您不能在 Visual Studio IDE 中设置此选项。有关详细信息，请参阅本主题前面的“默认值”或常用的 MSBuild 项目属性。

请参阅

C# Compiler Options

MSBuild 属性

/target (C# Compiler Options)

/target 编译器选项可以指定为以下四种形式之一：

/target:appcontainerexe

创建 Windows 8.x 应用商店 应用的 .exe 文件。

/target:exe

创建 .exe 文件。

/target:library

创建代码库。

/target:module

创建模块。

/target:winexe

创建 Windows 程序。

/target:winmdobj

创建一个 .winmdobj 中间文件。

如果不指定 **/target:module**，**/target** 会将 .NET Framework 程序集清单放入输出文件中。有关更多信息，请参见公共语言运行时中的程序集和公共特性。

程序集清单放置在编译中的第一个 .exe 输出文件中，如果没有 .exe 输出文件，会放置在第一个 DLL 中。例如，在以下的命令行中，清单将放置在 1.exe 中：

```
csc /out:1.exe t1.cs /out:2.netmodule t2.cs
```

编译器每次编译只创建一个程序集清单。关于编译中所有文件的信息全放在程序集清单中。除用 **/target:module** 创建的文件之外，所有输出文件都可以包含程序集清单。在命令行生成多个输出文件时，只能创建一个程序集清单，且必须放置在命令行上指定的第一个输出文件中。无论第一个输出文件是什么（**/target:exe**、**/target:winexe**、**/target:library** 或 **/target:module**），在同一编译中生成的任何其他输出文件都必须是模块 (**/target:module**)。

如果创建了一个程序集，则可以用 CLSCompliantAttribute 特性指示全部或部分代码是符合 CLS 的。

```
// target_clscompliant.cs
[assembly:System.CLSCompliant(true)]   // specify assembly complian

[System.CLSCompliant(false)]   // specify compliance for an element
public class TestClass
{
    public static void Main() {}
}
```

有关以编程方式设置此编译器选项的更多信息，请参见 OutputType。

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/subsystemversion (C# Compiler Options)

/target:appcontainerexe（C# 编译器选项）

如果使用 **/target:appcontainerexe** 编译器选项，则编译器会创建一个必须在应用容器中运行的 Windows 可执行 (.exe) 文件。此选项与 /target:winexe 等效，但专门用于 Windows 8.x 应用商店 应用。

语法

```
/target:appcontainerexe
```

备注

为了要求应用在应用容器中运行，此选项在可移植可执行 (PE) 文件中设置了一位。设置该位时，如果 CreateProcess 方法尝试在应用容器外启动该可执行文件，就会发生错误。

除非使用 /out 选项，否则输出文件名采用包含 Main 方法的输入文件的名称。

如果在命令提示符处指定此选项，则在下一个 **/out** 或 **/target** 选项之前，会使用所有文件来创建可执行文件。

在 IDE 中设置此编译器选项

1. 在“解决方案资源管理器”中，打开项目的快捷菜单，然后选择“属性”。
2. 在“应用程序”选项卡上，在“输出类型”列表中选择“**Windows** 应用商店应用”。

   此选项仅可用于 Windows 8.x 应用商店 应用模板。

有关如何以编程方式设置此编译器选项的信息，请参阅 OutputType。

以下命令将 filename.cs 编译为一个只能在应用容器中运行的 Windows 可执行文件。

```
csc /target:appcontainerexe filename.cs
```

请参阅

/target (C# Compiler Options)

/target:winexe (C# Compiler Options)

C# Compiler Options

/target:exe (C# Compiler Options)

/target:exe 选项使编译器创建可执行 (EXE) 的控制台应用程序。

语法

```
/target:exe
```

备注

默认情况下，**/target:exe** 选项有效。将创建带扩展名 .exe 的可执行文件。

使用 /target:winexe 创建可执行的 Windows 程序。

除非另外使用 /out 选项指定，否则输出文件名采用包含 Main 方法的输入文件名。

在命令行上指定该选项时，下一个 **/out** 或 **/target:module** 选项之前的所有文件都将用于创建 .exe 文件

在编译到 .exe 文件中的源代码文件中要求有且仅有一个 **main** 方法。 /main 编译器选项使您可以在代码中包含多个具有 **main** 方法的类的情况下，指定包含 **main** 方法的类。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“应用程序”属性页。
3. 修改“输出类型”属性。

有关如何以编程方式设置此编译器选项的信息，请参见 OutputType。

下列每个命令行都编译 in.cs，创建 in.exe：

```
csc /target:exe in.cs
csc in.cs
```

请参阅

/target (C# Compiler Options)

C# Compiler Options

/target:library (C# Compiler Options)

/target:library 选项使编译器创建一个动态链接库 (DLL) 而不是一个可执行文件 (EXE)。

语法

```
/target:library
```

备注

DLL 创建后会带有 .dll 扩展名。

除非使用 /out 选项另外指定，否则输出文件的名称采用第一个输入文件的名称。

在命令行上指定该选项时，下一个 **/out** 或 **/target:module** 选项之前的所有文件都将用于创建 .dll 文件。

生成 .dll 文件时，不需要 main 方法。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“应用程序”属性页。
3. 修改“输出类型”属性。

有关如何以编程方式设置此编译器选项的信息，请参见 OutputType。

编译 in.cs，创建 in.dll：

```
csc /target:library in.cs
```

请参阅

/target (C# Compiler Options)

C# Compiler Options

/target:module (C# Compiler Options)

此选项导致编译器不会生成程序集清单。

语法

```
/target:module
```

备注

默认情况下，使用此选项编译时所创建的输出文件具有 .netmodule扩展名。

没有程序集清单的文件无法由 .NET Framework 公共语言运行时加载。但可以通过 /addmodule 将这类文件并入程序集清单中。

如果在一次编译中创建了多个模块，一个模块中的 internal 类型可用于此编译中的其他模块。如果一个模块中的代码引用另一个模块中的 **internal** 类型，则两个模块都必须通过 **/addmodule** 合并到一个程序集清单中。

Visual Studio 开发环境不支持创建模块。

有关如何以编程方式设置此编译器选项的信息，请参见 OutputType。

编译 in.cs，创建 in.netmodule：

```
csc /target:module in.cs
```

请参阅

/target (C# Compiler Options)

C# Compiler Options

/target:winexe (C# Compiler Options)

/target:winexe 选项使编译器创建可执行 (EXE) 的 Windows 程序。

语法

```
/target:winexe
```

备注

将创建带扩展名 .exe 的可执行文件。Windows 程序是一种由 .NET Framework 库或使用 Win32 API 提供一个用户界面的程序。

使用 /target:exe 创建控制台应用程序。

除非另外使用 /out 选项指定，否则输出文件名采用包含 Main 方法的输入文件名。

在命令行指定该选项时，直至下一个 **/out** 或 /target 选项之前的所有文件都将用于创建 Windows 程序。

在编译到 .exe 文件中的源代码文件中要求有且仅有一个 **main** 方法。 /main 选项使您可以在代码中包含多个具有 **main** 方法的类的情况下，指定包含 **main** 方法的类。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“应用程序”属性页。
3. 修改“输出类型”属性。

有关如何以编程方式设置此编译器选项的信息，请参见 OutputType。

将 in.cs 编译成 Windows 程序：

```
csc /target:winexe in.cs
```

请参阅

/target (C# Compiler Options)

C# Compiler Options

/target:winmdobj（C# 编译器选项）

如果使用 **/target:winmdobj** 编译器选项，则编译器会创建一个中间 .winmdobj 文件，您可以将这个文件转换为 Windows 运行时二进制 (.winmd) 文件。之后，除了托管语言程序外，JavaScript 和 C++ 程序也可以使用该 .winmd 文件。

语法

```
/target:winmdobj
```

备注

winmdobj 设置会向编译器发出信号，表示需要中间模块。作为响应，Visual Studio 会将 C# 类库编译为 .winmdobj 文件。随后，.winmdobj 文件可作为可作为 WinMDExp 导出工具的输入，生成 Windows 元数据 (.winmd) 文件。.winmd 文件既包含原始库的代码，也包括 JavaScript（或 C++）以及 Windows 运行时使用的 WinMD 元数据的代码。

使用 **/target:winmdobj** 编译器选项所编译的文件，只能用作 WimMDExp 导出工具的输入；不能直接引用 .winmdobj 文件自身。

除非使用 /out 选项，否则输出文件的名称采用第一个输入文件的名称。不需要使用 Main 方法。

如果在命令提示符处指定 /target:winmdobj 选项，则在指定下一个 **/out** 或 /target:module 选项之前，会使用所有文件来创建 Windows 程序。

在 Visual Studio IDE 中为 Windows 应用商店应用程序设置此编译器选项

1. 在“解决方案资源管理器”中，打开项目的快捷菜单，然后选择“属性”。
2. 选择“应用程序”选项卡。
3. 在“输出类型”列表中，选择“**WinMD** 文件”。

   “**WinMD** 文件”选项只能用于 Windows 8.x 应用商店 应用模板。

有关如何以编程方式设置此编译器选项的信息，请参阅 OutputType。

以下命令将 filename.cs 编译为一个中间 .winmdobj 文件。

```
csc /target:winmdobj filename.cs
```

请参阅

/target (C# Compiler Options)

C# Compiler Options

/unsafe (C# Compiler Options)

/unsafe 编译器选项允许对使用 unsafe 关键字的代码进行编译。

语法

```
/unsafe
```

备注

有关不安全代码的更多信息，请参见 不安全代码和指针（C# 编程指南）。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“生成”属性页。
3. 选中“允许不安全代码”复选框。

有关如何以编程方式设置此编译器选项的信息，请参见 AllowUnsafeBlocks。

编译不安全模式的 in.cs：

```
csc /unsafe in.cs
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/utf8output (C# Compiler Options)

/utf8output 选项使用 UTF-8 编码显示编译器输出。

语法

```
/utf8output
```

备注

在某些国际配置中，编译器输出无法在控制台上正确显示。在这些配置中，请使用 **/utf8output** 并将编译器输出重定向到文件。

此编译器选项在 Visual Studio 中不可用，且不能通过编程方式进行更改。

请参阅

C# Compiler Options

/warn (C# Compiler Options)

/warn 选项指定编译器要显示的警告等级。

语法

```
/warn:option
```

参数

option

您希望为编译显示的警告等级：较小的数字只显示高严重度的警告；数字越大，显示的警告越多。有效值为 0 到 4：

警告等级	含义
0	关闭所有警告消息的显示。
1	显示严重的警告消息。
2	显示等级 1 警告以及某些不太严重的警告，如关于隐藏类成员的警告。
3	显示等级 2 警告以及某些不太严重的警告，例如有关总是计算为 **true** 或 **false** 的表达式的警告。
4（默认）	显示所有等级 3 警告以及信息性警告。

备注

若要获得有关错误或警告的信息，可以在帮助索引中查找错误代码。有关获得错误或警告信息的其他方式，请参见C# Compiler Errors。

使用 /warnaserror 可将所有的警告都视为错误。可以使用 /nowarn 禁用某些警告。

/w 是 **/warn** 的缩写形式。

在 **Visual Studio** 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“生成”属性页。
3. 修改“警告等级”属性。

有关如何以编程方式设置此编译器选项的信息，请参见 WarningLevel。

编译 in.cs 并让编译器仅显示等级 1 警告：

```
csc /warn:1 in.cs
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/warnaserror (C# Compiler Options)

/warnaserror+ 选项将所有警告都视为错误

语法

```
/warnaserror[<U>+</U> | -][:warning-list]
```

备注

将一般报告为警告的任何消息都报告为错误，并且暂停生成过程（不生成输出文件）。

默认情况下启用 **/warnaserror-**，这导致警告不会妨碍生成输出文件。**/warnaserror** 与 **/warnaserror+** 相同，它使警告被视为错误。

（可选）如果您希望只将几个特定的警告视为错误，可以指定一个以逗号分隔的列表，其中列出被视为错误的警告编号。

使用 /warn 可指定您希望编译器显示的警告等级。可以使用 /nowarn 禁用某些警告。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“生成”属性页。
3. 修改“将警告视为错误”属性。

   若要以编程方式设置此编译器选项，请参见 TreatWarningsAsErrors。

编译 in.cs 并且让编译器不显示警告：

```
csc /warnaserror in.cs
csc /warnaserror:642,649,652 in.cs
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/win32icon (C# Compiler Options)

/win32icon 选项在输出文件中插入 .ico 文件，这样.赋予输出文件在 Windows 资源管理器中的所需外观。

语法

```
/win32icon:filename
```

参数

filename

要添加到输出文件的 .ico 文件。

备注

可以使用资源编译器创建.ico 文件。编译 Visual C++ 程序时将调用资源编译器；.ico 文件是从 .rc 文件创建的。

请参见 /linkresource（用于引用 .NET Framework 资源文件）或 /resource（用于附加 .NET Framework 资源文件）。要导入 .res 文件，请参见 /win32res。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“应用程序”属性页。
3. 修改“应用程序图标”属性。

有关如何以编程方式设置此编译器选项的信息，请参见 ApplicationIcon。

编译 in.cs，并附加 .ico 文件 rf.ico 以生成 in.exe：

```
csc /win32icon:rf.ico in.cs
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

/win32manifest (C# Compiler Options)

使用 **/win32manifest** 选项可以指定要嵌入到项目的可迁移可执行 (PE) 文件中的用户定义的 Win32 应用程序清单文件。

语法

```
/win32manifest: filename
```

参数

filename

自定义清单文件的名称和位置。

备注

默认情况下，Visual C# 编译器嵌入指定“asInvoker”的请求执行级别的应用程序清单。它在生成该可执行文件的文件夹中创建清单，如果使用 Visual Studio，该文件夹通常为 bin\Debug 或 bin\Release。如果要提供自定义清单（例如，指定“highestAvailable”或“requireAdministrator”的请求执行级别的清单），请使用此选项指定文件名。

> 注意
>
> 此选项和 /win32res (C# Compiler Options) 选项是互斥的。如果尝试在同一命令行中同时使用这两个选项，则会收到一个生成错误。

如果应用程序没有用于指定请求执行级别的应用程序清单，则会受到 Windows Vista 的用户帐户控制功能下的文件/注册表虚拟化的影响。有关虚拟化的更多信息，请参见 The Windows Vista Developer Story: Windows Vista Application Development Requirements for User Account Control (UAC)。

如果满足下列任一条件，则应用程序会受到虚拟化的影响：

- 使用 **/nowin32manifest** 选项，并且在随后的生成步骤中未提供清单，或者没有通过使用 **/win32res** 选项将其包含在 Windows 资源 (.res) 文件中。
- 提供的自定义清单未指定请求执行级别。

Visual Studio 创建默认的 .manifest 文件，并将该文件与可执行文件一起存储在 debug 和 release 目录中。可以用任意文本编辑器创建一个清单，然后将该文件添加到项目中，以此添加自定义清单。或者，也可以右击“解决方案资源管理器”中

的“项目”图标，单击“添加新项”，然后单击“应用程序清单文件”。添加完新的或现有清单文件后，该文件将显示在“清单”下拉列表中。有关详细信息，请参阅“项目设计器”->“应用程序”页 (C#)。

提供应用程序清单的操作，可以作为自定义后期生成步骤，也可以通过使用 /nowin32manifest (C# Compiler Options) 选项作为 Win32 资源文件的组成部分。如果希望应用程序受到 Windows Vista 的文件/注册表虚拟化的影响，请使用该选项。这会阻止编译器在可迁移可执行 (PE) 文件中创建和嵌入默认清单。

下面的示例演示 Visual C# 编译器插入到 PE 中的默认清单。

> 注意
>
> 编译器将一个标准应用程序名称“MyApplication.app”插入到 xml 中。这是一种使应用程序可以在 Windows Server 2003 Service Pack 3 上运行的解决方法。

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion=
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3'
        <requestedExecutionLevel level="asInvoker"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

请参阅

C# Compiler Options

/nowin32manifest (C# Compiler Options)

如何：修改项目属性和配置设置

/win32res (C# Compiler Options)

/win32res 选项在输出文件中插入 Win32 资源。

语法

```
/win32res:filename
```

参数

filename

要添加到输出文件的资源文件。

备注

Win32 资源文件可以用资源编译器创建。在编译 Visual C++ 程序时会调用资源编译器；.res 文件是用 .rc 文件创建的。

Win32 资源可以包含版本或位图（图标）信息，这些信息有助于在文件资源管理器中标识您的应用程序。如果不指定 **/win32res**，编译器将根据程序集版本生成版本信息。

请参见 /linkresource（用于引用 .NET Framework 资源文件）或 /resource（用于附加 .NET Framework 资源文件）。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的“属性”页。
2. 单击“应用程序”属性页。
3. 单击“资源文件”按钮，然后使用组合框选择文件。

编译 in.cs，并附加 Win32 资源文件 rf.res 以生成 in.exe：

```
csc /win32res:rf.res in.cs
```

请参阅

C# Compiler Options

如何：修改项目属性和配置设置

C# Compiler Errors

某些 C# 编译器错误有相应的主题说明错误生成的原因，在某些情况下还说明了如何修复错误。若要了解某个特定错误消息是否有帮助信息，请使用以下步骤之一。

- 查看错误号 (例如 CS0029)，在 “输出”窗口为，然后搜索 MSDN。
- 选择该错误号 (例如 CS0029)，在 “输出”窗口，然后选择 F1 键。
- 在“索引”中的“查找”框中键入错误号。

如果这些步骤都不会产生有关该错误的信息，请转到结尾此页，然后发送包含错误数或文本的反馈。

有关在 C# 中如何配置错误和警告选项的信息，请参见“项目设计器”->“生成”页 (C#)。

注意
以下说明中的某些 Visual Studio 用户界面元素在计算机上出现的名称或位置可能会不同。 这些元素取决于你所使用的 Visual Studio 版本和你所使用的设置。 有关详细信息，请参阅个性化 Visual Studio IDE。

请参阅

C# Compiler Options

Sorry, we don't have specifics on this C# error

“项目设计器”->“生成”页 (C#)

/warn (C# Compiler Options)

/nowarn (C# Compiler Options)

Compiler Error CS0001

内部编译器错误

尝试确定编译器是否由于无法分析意外的语法而失败。如果重复收到此错误，请与 Microsoft 联系。

Compiler Error CS0006

未能找到元数据文件“<DLL 名称>”

编译了程序并给它显式传递了包含元数据的文件的名称，但没有找到 .dll。有关更多信息，请参见 /reference (C# Compiler Options)。

Compiler Error CS0007

意外的公共语言运行时初始化错误 —“description”

如果未能加载运行时，则会发生此错误。如果计算机上不存在编译器尝试加载的公共语言运行时的版本，或者公共语言运行时的安装或配置已损坏，则可能发生此错误。

如果更改了 csc.exe.config 文件，则可能发生上述情况。此文件是在安装期间配置的，不应更改。如果 csc.exe.config 文件有可能已被更改，请检查该文件，确保计算机上存在该文件中指定的运行时的版本。如果存在正确的版本，它可能已损坏。请重新安装公共语言运行时。

编译器错误 CS0015

类型“type”的名称太长

用户定义类型的完全限定名必须少于 1024 个字符，包括句点。

下面的示例生成 CS0015：

```
// CS0015.cs

// Remove a C from one of the namespace names or the class name to
// reducing the number of characters in the qualified class name to

namespace CC
{
    // The following namespace name has 510 characters.
    namespace CCC510charsCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
    {
        // The following class name has 510 characters. The qualifi
        // CC.CCC510chars…C.CCCC510chars…C (namespace.namespace.cla
        // characters, which causes compiler error CS0015.
        public class CCCC510charsCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
        {
            public static void Main()
            {
            }
        }
    }
}
```

Compiler Error CS0016

未能写入输出文件“<文件>”，原因是“<原因>”

编译器未能写入输出文件。检查文件的路径，确保文件存在。如果该位置已经存在一个以前生成的文件，请确保该文件可写入，并且当前没有进程将该文件锁定。例如，确保在尝试生成时可执行文件没有运行。

Compiler Error CS0019

运算符“operator”无法应用在“type”和“type”类型的操作数

对不支持二元运算符的数据类型应用该运算符。例如，您不能对字符串使用 || 运算符，不能对 bool 变量使用 +、-、< 或 > 运算符，并且不能将 ==运算符与 **struct** 类型一起使用（除非该类型显式重载了该运算符）。

如果在处理类类型时遇到此错误，则原因是该类未重载运算符。有关详细信息，请参阅可重载运算符（C# 编程指南）。

在下面的示例中，将在两个位置生成 CS0019，原因是 C# 中的 bool 不能转换为 int。将减法运算符应用于字符串时，还会生成 CS0019。请注意，加法运算符 (+) 可与字符串操作数一起使用，因为 **String** 类会重载该运算符以执行字符串串联。

```
static void Main()
{
    bool result = true;
    if (result > 0) //CS0019
    {
        // Do something.
    }

    int i = 1;
    // You cannot compare an integer and a boolean value.
    if (i == true) //CS0019
    {
        //Do something...
    }

    // The following use of == causes no error. It is the compariso
    // an integer and a boolean value that causes the error in the
    // previous if statement.
    if (result == true)
    {
        //Do something...
    }

    string s = "Just try to subtract me.";
    float f = 100 - s; // CS0019
}
```

在下面的示例中，必须在 ConditionalAttribute 外指定条件逻辑。只能向 ConditionalAttribute 传递一个预定义符号。

下面的示例生成 CS0019。

```
// CS0019_a.cs
// compile with: /target:library
using System.Diagnostics;
public class MyClass
{
   [ConditionalAttribute("DEBUG" || "TRACE")]   // CS0019
   public void TestMethod() {}

   // OK
   [ConditionalAttribute("DEBUG"), ConditionalAttribute("TRACE")]
   public void TestMethod2() {}
}
```

请参阅

运算符（C# 编程指南）

隐式数值转换表（C# 参考）

Compiler Error CS0029

无法将类型“type”隐式转换为“type”

编译器要求显式转换。例如，可能需要将右值转换成与左值相同的类型。或者必须提供转换例程以支持某些运算符重载。

在将某个类型的变量赋给其他类型的变量时必须进行转换。当在不同类型的变量之间进行赋值时，编译器必须将赋值运算符右边的类型转换为该赋值运算符左边的类型。假设有下面的代码：

```
int i = 50;
long lng = 100;
i = lng;
```

i = lng; 进行赋值运算，但赋值运算符左右两边变量的数据类型不匹配。进行赋值前，编译器将变量 lng（类型为 long）隐式转换为 int。此为隐式转换，原因是没有代码显式指示编译器执行此转换。此代码的问题在于上述转换被视为收缩转换，而编译器不允许进行隐式收缩转换，原因是可能会丢失数据。

如果转换后的数据类型所占用的内存存储空间比转换前的数据类型所占用的少，则存在收缩转换。例如，将 long 类型转换为 int 类型就被视为收缩转换。long 类型占用 8 个字节的内存，而 int 类型只占用 4 个字节。若要查看数据丢失如何发生，请考虑以下示例：

```
int i = 50;
long lng = 3147483647;
i = lng;
```

变量 lng 现在包含的值无法存储在变量 i 中，原因是该值太大。如果要将该值转换为 int 类型，就会丢失一些数据，并且转换后的值不同于转换前的值。

扩大转换与收缩转换相反。对于扩大转换，转换后的数据类型占用的内存存储空间比转换前的数据类型占用的多。下面是一个扩大转换的示例：

```
int i = 50;
long lng = 100;
lng = i;
```

请注意此代码示例和第一个示例之间的区别。这里，变量 lng 位于赋值运算符的左边，所以它是赋值的目标。在可以进行赋值前，编译器必须将变量 i（类型为 int）隐式转换为 long 类型。这是一个扩大转换，因为我们是从一个占用 4 个字节内存

的类型 (int) 转换为一个占用 8 个字节内存的类型 (long)。由于不会发生数据丢失，所以允许进行隐式扩大转换。任何可以用 int 类型存储的值也可以用 long 类型存储。

我们知道隐式收缩转换是不允许的，因此为了能够编译这些代码，就需要显式转换数据类型。显式转换是使用强制转换来完成的。强制转换是 C# 中用来描述将一种数据类型转换为另一种数据类型的术语。若要编译这些代码，我们需要使用以下语法：

```
int i = 50;
long lng = 100;
i = (int) lng;   // cast to int
```

第三行代码通知编译器在进行赋值前，将变量 lng（类型为 long）显式转换为 int 类型。切记，使用收缩转换可能会丢失数据。使用收缩转换时应小心，而且即使可以编译代码，也可能会在运行时获得意外的结果。

此讨论只针对值类型。使用值类型就是直接使用存储在变量中的数据。但 .NET Framework 还具有引用类型。使用引用类型就是使用对变量的引用，而不是使用实际数据。引用类型的示例是类、接口和数组。不能隐式或显式地将一个引用类型转换为其他引用类型，除非编译器允许特定的转换或可以实现相应的转换运算符。

下面的示例生成 CS0029：

```
// CS0029.cs
public class MyInt
{
    private int x = 0;

    // Uncomment this conversion routine to resolve CS0029
    /*
    public static implicit operator int(MyInt i)
    {
       return i.x;
    }
    */

    public static void Main()
   {
      MyInt myInt = new MyInt();
      int i = myInt; // CS0029
   }
}
```

请参阅

转换运算符（C# 编程指南）

Compiler Error CS0034

运算符“operator”对于“type1”和“type2”类型的操作数具有二义性

运算符用在了两个对象上，并且编译器找到多个转换。因为转换必须是唯一的，因此必须进行一次强制转换，或者使其中一个转换成为显式转换。有关更多信息，请参见转换运算符（C# 编程指南）。

下面的示例生成 CS0034：

```
// CS0034.cs
public class A
{
   // allows for the conversion of A object to int
   public static implicit operator int (A s)
   {
      return 0;
   }

   public static implicit operator string (A i)
   {
      return null;
   }
}

public class B
{
   public static implicit operator int (B s)
   // one way to resolve this CS0034 is to make one conversion expl
   // public static explicit operator int (B s)
   {
      return 0;
   }

   public static implicit operator string (B i)
   {
      return null;
   }

   public static implicit operator B (string i)
   {
      return null;
   }

   public static implicit operator B (int i)
   {
      return null;
   }
}
```

```
public class C
{
   public static void Main ()
   {
      A a = new A();
      B b = new B();
      b = b + a;   // CS0034
      // another way to resolve this CS0034 is to make a cast
      // b = b + (int)a;
   }
}
```

在 C# 1.1 中，一个编译器器 Bug 使其能够定义一个类，在其中实现针对 **int** 和 **bool** 的用户定义的隐式转换，并对该类型的对象使用按位 **AND** 运算符 (**&**)。在 C# 2.0 中，已修复此 Bug 以使编译器符合 C# 规范，因此以下代码现在将导致错误 CS0034：

```
namespace CS0034
{
    class TestClass2
    {
        public void Test()
        {
            TestClass o1 = new TestClass();
            TestClass o2 = new TestClass();
            TestClass o3 = o1 & o2; //CS0034
        }
    }

    class TestClass
    {
        public static implicit operator int(TestClass o)
        {
            return 1;
        }

        public static implicit operator TestClass(int v)
        {
            return new TestClass();
        }

        public static implicit operator bool(TestClass o)
        {
            return true;
        }
    }

}
```

Compiler Error CS0038

无法通过嵌套类型“type2”来访问外部类型“type1”的非静态成员

类中的字段不自动可用于嵌套类。若要可用于嵌套类，该字段必须是 static。否则，必须创建外部类的实例。有关更多信息，请参见嵌套类型（C# 编程指南）。

下面的示例生成 CS0038：

```
// CS0038.cs
class OuterClass
{
   public int count;
   // try the following line instead
   // public static int count;

   class InnerClass
   {
      void func()
      {
         // or, create an instance
         // OuterClass class_inst = new OuterClass();
         // int count2 = class_inst.count;
         int count2 = count;   // CS0038
      }
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0039

无法通过引用转换、装箱转换、取消装箱转换、包装转换或 Null 类型转换将类型“type1”转换为“type2”

继承、引用转换和装箱转换允许使用 as（C# 参考） 运算符的转换。有关更多信息，请参见转换运算符（C# 编程指南）。

下面的示例生成 CS0039。

```
// CS0039.cs
using System;
class A
{
}
class B: A
{
}
class C: A
{
}
class M
{
    static void Main()
    {
        A a = new C();
        B b = new B();
        C c;

        // This is valid; there is a built-in reference
        // conversion from A to C.
        c = a as C;

        //The following generates CS0039; there is no
        // built-in reference conversion from B to C.
        c = b as C;  // CS0039
    }
}
```

Compiler Error CS0050

可访问性不一致：返回类型“type”比方法“method”的可访问性低

方法的返回类型和形参表中引用的每个类型都必须至少具有和方法自身相同的可访问性。有关更多信息，请参见访问修饰符（C# 编程指南）。

下面的示例生成 CS0050，因为没有为 MyClass 提供任何可访问性修饰符，因此它的可访问性默认为 **private**。

```
// CS0050.cs
class MyClass //accessibility defaults to private
// try the following line instead
// public class MyClass
{
}

public class MyClass2
{
    public static MyClass MyMethod()   // CS0050
    {
        return new MyClass();
    }

    public static void Main() { }
}
```

Compiler Error CS0051

可访问性不一致：参数类型“type”比方法“method”的访问性低

方法的返回类型和形参表中引用的每个类型都必须至少具有和方法自身相同的可访问性。请确保方法签名中使用的类型不会因为省略 **public** 修饰符而意外变为专用。有关更多信息，请参见访问修饰符（C# 编程指南）。

下面的示例生成 CS0051：

```
// CS0051.cs
public class A
{
    // Try making B public since F is public
    // B is implicitly private here
    class B
    {
    }

    public static void F(B b)  // CS0051
    {
    }

    public static void Main()
    {
    }
}
```

Compiler Error CS0052

可访问性不一致: 字段类型“type”比字段“field”的访问性低

字段类型的可访问性不能比字段本身的可访问性低，因为所有的公共构造都必须返回公共的可访问对象。

下面的示例生成 CS0052：

```
// CS0052.cs
public class MyClass2
{
    // The following line causes an error because the field, M, is
    // as public, but the type, MyClass, is private. Therefore the
    // less accessible than the field.
    public MyClass M;   // CS0052

    private class MyClass
    {
    }
    // One way to resolve the error is to change the accessibility
    // to public.
    //public class MyClass
    // Another solution is to change the accessibility of the fiel
    // private MyClass M;
}

    public class MainClass
    {
        public static void Main()
        {
        }
    }
```

请参阅

C# 关键字

访问修饰符（C# 参考）

可访问性级别（C# 参考）

修饰符（C# 参考）

Compiler Error CS0071

事件的显式接口实现必须使用事件访问器语法

当显式实现在接口中声明的事件时，您必须手动提供通常由编译器提供的 **add** 和 **remove** 事件访问器。访问器代码可将类中的接口事件连接到另一事件（稍后在本主题中演示），或连接到其自身的委托类型。有关更多信息，请参见如何：实现接口事件（C# 编程指南）。

下面的示例生成 CS0071。

```
// CS0071.cs
public delegate void MyEvent(object sender);

interface ITest
{
    event MyEvent Clicked;
}

class Test : Itest
{
    event MyEvent ITest.Clicked;  // CS0071

    // try the following code instead
/*
private MyEvent clicked;

    event MyEvent Itest.Clicked
    {
        add
        {
            clicked += value;
        }
        remove
        {
            clicked -= value;
        }
    }
*/
    public static void Main() { }
}
```

Compiler Error CS0103

当前上下文中不存在名称“identifier”

尝试使用类、命名空间或范围中不存在的名称。检查名称的拼写，并检查 正在使用的目录和程序集引用，确保您尝试使用的名称可用。

如果在循环或者 **try** 或 **if** 块中声明一个变量，然后尝试从封闭代码块或一独立代码块访问此变量，经常会发生此错误。如下面的示例所示。

```
using System;

class MyClass1
{
    public static void Main()
    {
        try
        {
            // The following declaration is only available inside t
            MyClass1 conn = new MyClass1();
        }
        catch (Exception e)
        {
            // The following expression causes error CS0103, becaus
            // conn only exists in the try block.
            if (conn != null)
                Console.WriteLine("{0}", e);
        }
    }
}
```

下面的示例解决错误。

```
using System;

class MyClass2
{
    public static void Main()
    {
        // To resolve the error in the example, the first step is t
        // move the declaration of conn out of the try block. The f
        // declaration is available throughout the Main method.
        MyClass2 conn = null;
        try
        {
            // Inside the try block, use the conn variable that you
            // previously.
            conn = new MyClass2();
        }
        catch (Exception e)
        {
            // The following expression no longer causes an error,
            // the declaration of conn is in scope.
            if (conn != null)
                Console.WriteLine("{0}", e);
        }
    }
}
```

Compiler Error CS0106

修饰符“modifier”对该项无效

类或接口成员是用无效的访问修饰符标记的。下列示例描述了一些无效的修饰符：

- 在接口方法上不允许使用 static 和 public 修饰符。
- 在显式接口声明上不允许使用 **public** 关键字。在这种情况下，请从显式接口声明中移除 **public** 关键字。
- 在显式接口声明上不允许使用 abstract 关键字，因为显式接口实现永远不能被重写。

在以前的 Visual Studio 版本中，不允许在类上使用 **static** 修饰符，但允许 **static** 类以 Microsoft Visual Studio 2005 开头。

有关更多信息，请参见 接口（C# 编程指南）

下面的示例生成 CS0106。

```
// CS0106.cs
namespace MyNamespace
{
   interface I
   {
      void m();
      static public void f();   // CS0106
   }

   public class MyClass
   {
      public void I.m() {}   // CS0106
      public static void Main() {}
   }
}
```

Compiler Error CS0115

“function”：没有找到适合的方法来重写

方法被标记为 override，但编译器未找到可重写的方法。有关更多信息，请参见 override（C# 参考）和了解何时使用 Override 和 New 关键字（C# 编程指南）。

下面的示例生成 CS0115。可以用下列两种方法之一解决 CS0115：

- 从 MyClass2 的方法中移除 **override** 关键字。
- 将 MyClass1 用作 MyClass2 的基类。

```
// CS0115.cs
namespace MyNamespace
{
    abstract public class MyClass1
    {
        public abstract int f();
    }

    abstract public class MyClass2
    {
        public override int f()   // CS0115
        {
            return 0;
        }

        public static void Main()
        {
        }
    }
}
```

Compiler Error CS0116

命名空间并不直接包含诸如字段或方法之类的成员

在 namespace 内，编译器只接受类、结构、联合、枚举、接口和委托。当 C/C++ 开发人员忘记了在 C# 中，方法和变量必须在结构或类中定义时，通常会生成此错误。有关更多信息，请参见C# 程序的通用结构（C# 编程指南）。

下面的示例生成 CS0116：

```
// CS0116.cs
namespace x
{
    using System;

    // method must be in class/struct
    void Method(string str) // CS0116
    {
        Console.WriteLine(str);
    }
    // To fix the error, you must
    // enclose a method in a class:
    class Program
    {
        void Method2(string str)
        {
            Console.WriteLine(str);
        }
    }
}
```

请参阅

C# 程序的通用结构（C# 编程指南）

类和结构（C# 编程指南）

命名空间（C# 编程指南）

Compiler Error CS0120

非静态的字段、方法或属性“member”要求对象引用

必须首先创建对象实例，才能使用非静态的字段、方法或属性。有关静态方法的更多信息，请参见静态类和静态类成员（C# 编程指南）。有关创建类的实例的更多信息，请参见实例构造函数（C# 编程指南）。

下面的示例生成 CS0120：

```
// CS0120_1.cs
public class MyClass
{
   // Non-static field
   public int i;
   // Non-static method
   public void f(){}
   // Non-static property
   int Prop
   {
      get
      {
         return 1;
      }
   }

   public static void Main()
   {
      i = 10;   // CS0120
      f();   // CS0120
      int p = Prop;   // CS0120
      // try the following lines instead
      // MyClass mc = new MyClass();
      // mc.i = 10;
      // mc.f();
      // int p = mc.Prop;
   }
}
```

如果从静态方法调用非静态方法，也会生成 CS0120，如下所示：

```
// CS0120_2.cs
// CS0120 expected
using System;

public class MyClass
{
   public static void Main()
   {
      TestCall();   // CS0120
      // To call a non-static method from Main,
      // first create an instance of the class.
      // Use the following two lines instead:
      // MyClass anInstanceofMyClass = new MyClass();
      // anInstanceofMyClass.TestCall();
   }

   public void TestCall()
   {
   }
}
```

同样，静态方法不能调用实例方法，除非显式给它提供了类的实例，如下所示：

```
// CS0120_3.cs
using System;

public class MyClass
{
   public static void Main()
   {
      do_it("Hello There");   // CS0120
   }

   private void do_it(string sText)
   // You could also add the keyword static to the method definitio
   // private static void do_it(string sText)
   {
      Console.WriteLine(sText);
   }
}
```

请参阅

[类和结构（C# 编程指南）]

Compiler Error CS0122

不可访问“member”，因为它受保护级别限制

类成员的访问修饰符禁止访问该成员。有关更多信息，请参见访问修饰符（C# 编程指南）。

出现此错误（未在下面的示例中显示）的一个原因是：在友元程序集的目标上省略了 **/out** 编译器标志。有关更多信息，请参见 友元程序集（C# 和 Visual Basic） 和 /out (C# Compiler Options)

下面的示例生成 CS0122：

```
// CS0122.cs
public class MyClass
{
    // Make public to resolve CS0122
    void MyMethod()
    {
    }
}

public class MyClass2
{
    public static int Main()
    {
        MyClass a = new MyClass();
        // MyMethod is private
        a.MyMethod();   // CS0122
        return 0;
   }
}
```

Compiler Error CS0134

“variable”的类型为“type”。只能用 null 对引用类型(字符串除外)的常量字段进行初始化。

常量表达式是可在编译时可完全计算的表达式。由于应用 new 运算符是创建引用类型的非空值的唯一方法，并且常量表达式中不允许使用 new 运算符，因此除字符串外，引用类型常量的唯一可能值为 null。

如果尝试创建 const 字符串数组时遇到此错误，则解决方案是使该数组成为只读数组，并在构造函数中将其初始化。

下面的示例生成 CS0134。

```
// CS0134.cs
// compile with: /target:library
class MyTest {}

class MyClass
{
   const MyTest test = new MyTest();   // CS0134

   //OK
   const MyTest test2 = null;
   const System.String test3 = "test";
}
```

Compiler Error CS0151

应输入整型值

在需要整型数据类型的情况中使用了变量。有关更多信息，请参见类型（C# 编程指南）。

当没有转换或者可用的隐式转换导致了不明确的情形时，会出现此错误。下面的示例生成 CS0151。

```
// CS0151.cs
public class MyClass
{
   public static implicit operator int (MyClass aa)
   {
      return 0;
   }

   public static implicit operator long (MyClass aa)
   {
      return 0;
   }

   public static void Main()
   {
      MyClass a = new MyClass();

      // Compiler cannot choose between int and long
      switch (a)   // CS0151
      // try the following line instead
      // switch ((int)a)
      {
         case 1:
            break;
      }
   }
}
```

在 Visual Studio 2008 及更高版本中，void 方法调用会生成 CS0151。通过调用返回整数类型（如 int 或 long）的方法可以修复该错误。

```
class C
{
    static void Main()
    {

        switch (M()) // CS0151
        {
            default:
                break;
        }
    }

    static void M()
    {
    }
}
```

编译器错误 CS0163

控制不能从一个 case 标签("label")贯穿到另一个 case 标签

当switch 语句 包含多个时开关部分，使用以下关键字之一，必须显式终止每节，其中包括最后一个：

- return
- goto
- break
- throw
- continue

如果要由一部分到下一部分实现"贯穿"的行为，使用 goto case #。有关更多信息和示例，请参见switch（C# 参考）。

下面的示例生成 CS0163。

```
// CS0163.cs
public class MyClass
{
    public static void Main()
    {
        int i = 0;

        switch (i)   // CS0163
        {
            // Compiler error CS0163 is reported on the following l
            case 1:
                i++;
            // To resolve the error, uncomment one of the following
            // return;
            // break;
            // goto case 3;

            case 2:
                i++;
                return;

            case 3:
                i = 0;
                return;

            // Compiler error CS0163 is reported on the following l
            default:
                Console.WriteLine("Default");
                // To resolve the error, uncomment the following li
            //break;
        }
    }
```

Compiler Error CS0165

使用了未赋值的局部变量“name”

C# 编译器不允许使用未初始化的变量。如果编译器检测到使用了可能未初始化的变量，就会生成 编译错误CS0165。有关详细信息，请参阅字段（C# 编程指南）。请注意，当编译器遇到可能会导致使用未赋值变量的构造时，即使您的特定代码不会使用该变量，也会生成此错误。这样可以避免对明确赋值使用过度复杂的规则。

有关更多信息，请参见 http://blogs.msdn.com/ericlippert/archive/2006/08/18/706398.aspx。

下面的示例生成 CS0165：

```
// CS0165.cs
using System;

class MyClass
{
    public int i;
}

class MyClass2
{
    public static void Main(string[] args)
    {
        // i and j are not initialized.
        int i, j;

        // You can provide a value for args[0] in the 'Command line
        // text box on the Debug tab of the project Properties wind
        if (args[0] == "test")
        {
            i = 0;
        }
        // If the following else clause is absent, i might not be
        // initialized.
        //else
        //{
        //    i = 1;
        //}

        // Because i might not have been initialized, the following
        // line causes CS0165.
        j = i;

        // To resolve the error, uncomment the else clause of the p
        // if statement, or initialize i when you declare it.

        // The following example causes CS0165 because myInstance i
        // declared but not instantiated.
        MyClass myInstance;
        // The following line causes the error.
        myInstance.i = 0;

        // To resolve the error, replace the previous declaration w
        // the following line.
        //MyClass myInstance = new MyClass();
    }
}
```

编译器错误 CS0165 在递归委托定义来生成。可以通过在以下两个语句中定义委托可避免此错误，以便在变量初始化之前，不使用变量。下面的示例演示错误和解决方案。

```
class Program
{
    delegate void Del();
    static void Main(string[] args)
    {
        // The following line causes CS0165 because variable d is u
        // as an argument before it has been initialized.
        Del d = delegate() { System.Console.WriteLine(d); };

        //// To resolve the error, initialize d in a separate state
        //Del d = null;
        //// After d is initialized, you can use it as an argument.
        //d = delegate() { System.Console.WriteLine(d); };
        //d();
    }
}
```

Compiler Error CS0173

无法确定条件表达式的类型，因为“class1”和“class2”之间没有隐式转换

当希望不同类的对象在同一代码中使用时，类之间的转换非常有用。然而，在一起工作的两个类不能有相互转换和多余转换，也不能有隐式转换。 *class1* 和 *class2* 的类型分别确定，选择更全面的类型作为条件表达式的类型。有关类型的确定方式的更多信息，请参见Conditional operator cannot cast implicitly。

若要解决 CS0173，请确认 *class1* 与 *class2* 之间有且仅有一个隐式转换，而不论向哪个方向进行转换或在哪个类中进行转换。有关更多信息，请参见隐式数值转换表（C# 参考）和转换运算符（C# 编程指南）。

下面的示例生成 CS0173：

```
// CS0173.cs
public class C {}

public class A
{
    //// The following code defines an implicit conversion operator
    //// type C to type A.
    //public static implicit operator A(C c)
    //{
    //    A a = new A();
    //    a = c;
    //    return a;
    //}
}

public class MyClass
{
    public static void F(bool b)
    {
        A a = new A();
        C c = new C();

        // The following line causes CS0173 because there is no imp
        // conversion from a to c or from c to a.
        object o = b ? a : c;

        // To resolve the error, you can cast a and c.
        //object o = b ? (object)a : (object)c;

        // Alternatively, you can add a conversion operator from cl
        // class A, or from class A to class C, but not both.
    }

   public static void Main()
   {
      F(true);
   }
}
```

下面的代码在 Visual Studio 2005 中不生成 CS0173，但在更高版本中会生成 CS0173。

```
//cs0173_2.cs
class M
{
    static int Main ()
    {
        int X = 1;
        // The following line causes CS0173 in Visual Studio 2005.
        object o = (X == 0) ? null : null;
        return -1;
    }
}
```

Compiler Error CS0178

无效的秩说明符：应为“,”或“]”

数组初始化的格式错误。例如，在指定数组维数时，可以指定以下内容：

- 中括号中的数字。
- 空的中括号。
- 括在中括号中的逗号。

有关更多信息，请参见 数组（C# 编程指南） 和 C# 规范 (C# 语言规范) 中有关数组初始值设定项的部分。

下面的示例生成 CS0178。

```
// CS0178.cs
class MyClass
{
   public static void Main()
   {
      int a = new int[5][,][][5;    // CS0178
      int[,] b = new int[3,2];    // OK

      int[][] c = new int[10][];
      c[0] = new int[5][5];    // CS0178
      c[0] = new int[2];    // OK
      c[1] = new int[2]{1,2};    // OK
   }
}
```

Compiler Error CS0188

在给“this”对象的所有字段赋值之前，无法使用该对象

构造函数必须给 **struct** 中的所有字段都赋值，然后才能调用 **struct** 中的方法。

如果在尝试初始化结构构造函数中的属性时看到此错误，则解决方案将更改此构造函数参数以指定支持字段而不指定该属性本身。由于自动实现的属性不具有支持字段，因此在任何情况下都不可能从构造函数进行初始化，从而应在结构中避免使用自动实现的属性。

有关更多信息，请参见使用结构（C# 编程指南）。

下面的示例生成 CS0188：

```
// CS0188.cs
// compile with: /t:library
namespace MyNamespace
{
    class MyClass
    {
        struct S
        {
            public int a;

            void MyMethod()
            {
            }

            S(int i)
            {
                // a = i;
                MyMethod();  // CS0188
            }
        }
        public static void Main()
        { }

    }
}
```

请参阅

结构（C# 编程指南）

自动实现的属性（C# 编程指南）

Compiler Error CS0201

只有 assignment、call、increment、decrement 和 new 对象表达式可用作语句

编译器在遇到无效语句时会生成错误。无效语句是以未表示以下各项的分号结尾的任意行或一系列行：赋值 (=)、方法调用 ()、new、-- 或 ++ 运算。有关更多信息，请参见语句、表达式和运算符（C# 编程指南）。

下面的示例将生成 CS0201，因为 2 * 3 是表达式，而不是语句。若要使代码能够编译，请尝试将表达式的值赋给变量。

```
// CS0201.cs
public class MainClass
{
   public static void Main()
   {
      2 * 3;   // CS0201
      // Try the following line instead.
      //   int i = 2 * 3;
   }
}
```

下面的示例将生成 CS0201，因为 checked 本身并不是语句，即使已通过递增操作对其进行了参数化。

```
// CS0201_b.cs
// compile with: /target:library
public class MyList<T>
{
   public void Add(T x)
   {
      int i = 0;
      if ( (object)x == null)
      {
         checked(i++);   // CS0201

         // OK
         checked {
            i++;
         }
      }
   }
}
```

请参阅

C# Compiler Errors

Compiler Error CS0229

“member1”和“member2”之间具有二义性

不同接口的成员具有相同的名称。如果要保留相同的名称，则必须限定这些名称。有关更多信息，请参见接口（C# 编程指南）。

注意
在某些情况下，此二义性可以通过使用 using 别名向标识符提供明确的前缀来解决。

下面的示例生成 CS0229：

```
// CS0229.cs

interface IList
{
    int Count
    {
        get;
        set;
    }

    void Counter();
}

interface Icounter
{
    double Count
    {
        get;
        set;
    }
}

interface IListCounter : IList , Icounter {}

class MyClass
{
    void Test(IListCounter x)
    {
        x.Count = 1;  // CS0229
        // Try one of the following lines instead:
        // ((IList)x).Count = 1;
        // or
        // ((Icounter)x).Count = 1;
    }

    public static void Main() {}
}
```

Compiler Error CS0233

“identifier”没有预定义大小，因此 sizeof 只能用于不安全的上下文中（请考虑使用 System.Runtime.InteropServices.Marshal.SizeOf）

sizeof 运算符只能用于为编译时常数的类型。如果您遇到此错误，请确保在编译时可以确定标识符的大小。如果不能确定，则用 SizeOf 代替 **sizeof**。

下面的示例生成 CS0233：

```
// CS0233.cs
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
public struct S
{
    public int a;
}

public class MyClass
{
    public static void Main()
    {
        S myS = new S();
        Console.WriteLine(sizeof(S));   // CS0233
        // Try the following line instead:
        // Console.WriteLine(Marshal.SizeOf(myS));
    }
}
```

Compiler Error CS0234

命名空间“namespace”中不存在类型或命名空间名称“name”（是否缺少程序集引用？）

应为某个类型。此错误可能的原因包括：

- 在编译中未引用包含某个类型定义的程序集；请使用 /reference（导入元数据）指定程序集
- 将某个变量名传递给了 typeof 运算符。
- 您尝试引用的程序集不属于目标 .NET Framework 配置文件的一部分。有关更多信息，请参见 .NET Framework 目标错误疑难解答。

如果在将代码从一台开发计算机转移到另一台开发计算机后看到此错误，请确保新计算机上的项目具有正确的引用，并且程序集的版本与旧计算机上的程序集版本相同。您也可以使用对象浏览器来检查程序集，并验证该程序集是否包含其预期要包含的类型。

下面的示例生成 CS0234：

```
// CS0234.cs
public class C
{
   public static void Main()
   {
      System.DateTime x = new System.DateTim();   // CS0234
      // try the following line instead
      // System.DateTime x = new System.DateTime();
   }
}
```

Compiler Error CS0246

未能找到类型或命名空间名称“type/namespace”（是否缺少 using 指令或程序集引用？）

未找到程序中使用的类型或命名空间。可能忘了引用 (/reference) 包含该类型的程序集，或者可能未用 using 指令正确地限定其用法。

以下情况会导致编译器错误 CS0246。

- 类型或命名空间的名称拼写是否有误？没有正确的名称，编译器就无法找到类型或命名空间的定义。这通常是由于类型名称中使用的大小写形式不正确。例如，Dataset ds; 会生成 CS0246，因为 Dataset 中的 s 必须大写。
- 如果错误的原因在于命名空间名称，您是否添加了对于包含该命名空间的程序集的引用 (/reference)？例如，您的代码可能包含指令 using Accessibility。但是，如果项目不引用程序集 Accessibility.dll，则会报告错误 CS0246。
- 如果错误的原因在于类型名称，您是否包括了正确的 using 指令，或是否已完全限定该类型的名称？考虑下列声明：DataSet ds。若要使用 **DataSet** 类型，需要具备两个条件。第一，需要一个对包含 **DataSet** 类型定义的程序集的引用。第二，需要对 **DataSet** 所在的命名空间使用 **using** 指令。例如，由于 **DataSet** 位于 **System.Data** 命名空间中，所以需要在代码的开头使用以下指令：using System.Data。

  using 指令不是必需的。但如果省略该指令，则在引用 **DataSet** 类型时必须对它进行完全限定。完全限定意味着每次在代码中引用类型时都要同时指定命名空间和类型。如果在上一示例中省略 **using** 指令，则必须编写 System.Data.DataSet ds 来声明 ds 而不是 DataSet ds。
- 您是否在需要类型的地方使用了变量或其他某种语言元素？例如，在 **is** 语句中，如果您使用 **Type** 对象而不是实际的类型，将会出现错误 CS0246。
- 是否使用了 *using* 别名指令，但没有完全限定类型名称？ **using** 别名指令不使用源代码文件中的 **using** 指令来解析类型。下面的示例生成 CS0246，因为类型 List<int> 未完全限定。 System.Collections.Generic 的 **using** 指令无法避免该错误。

  ```
  using System.Collections.Generic;

  // The following declaration generates CS0246.
  using myAliasName = List&lt;int&gt;;

  // To avoid the error, fully qualify List.
  using myAliasName2 = System.Collections.Generic.List&lt;int&gt;
  ```

下面的示例生成 CS0246，因为缺少必要的 **using** 指令。

```
// CS0246.cs
//using System.Diagnostics;

public class MyClass
{
    // The following line causes CS0246\. To fix the error, uncomme
    // the using directive for the namespace for this attribute,
    // System.Diagnostics.
    [Conditional("A")]
    public void Test()
    {
    }

    public static void Main()
    {
    }
}
```

下面的示例会引发 CS0246，因为在需要实际类型的位置使用了类型为 **Type** 的对象。

```
// CS0246b.cs
using System;

class ExampleClass
{
    public bool supports(object o, Type t)
    {
        // The following line causes CS0246\. You must use an
        // actual type, such as ExampleClass, String, or Type.
        if (o is t)
        {
            return true;
        }
        return false;
    }
}

class Program
{
    public static void Main()
    {
        ExampleClass myC = new ExampleClass();
        myC.supports(myC, myC.GetType());
    }
}
```

Compiler Error CS0260

类型“type”的声明上缺少分部修饰符；存在此类型的其他分部声明

此错误指示您声明具有相同名称的多个类。此外，至少一个，但并非全部声明包含一个 **partial** 修饰符。如果您想在若干部分中定义一个类，通过使用关键字 **partial**，必须声明每个部分。

如果您声明一个新类碰巧与同一命名空间中的其他位置声明的分部类具有相同的名称，则也会发生此错误。

下面的示例生成 CS0260：

```
// CS0260.cs
// You must mark both parts of the definition of class C
// by using the partial keyword.

// The following line causes CS0260\. To resolve the error, add
// the 'partial' keyword to the declaration.
class C
{
}

partial class C
{
}
```

请参阅

分部类和方法（C# 编程指南）

Compiler Error CS0266

无法将类型“type1”隐式转换为“type2”。存在显式转换（是否缺少强制转换？）

当代码尝试在无法进行隐式转换的两个类型上尝试转换时，此错误发生。有关详细信息，请参阅强制转换和类型转换（C# 编程指南）。

下面的代码演示生成 CS0266 的示例。

```
// CS0266.cs
class MyClass
{
    public static void Main()
    {
        // You cannot implicitly convert a double to an integer.
        double d = 3.2;

        // The following line causes compiler error CS0266.
        int i1 = d;

        // However, you can resolve the error by using an explicit
        int i2 = (int)d;

        // You cannot implicitly convert an object to a class type
        object obj = "MyString";

        // The following assignment statement causes error CS0266.
        MyClass myClass = obj;

        // You can resolve the error by using an explicit conversic
        MyClass c = ( MyClass )obj;

        // You cannot implicitly convert a base class object to a c
        MyClass mc = new MyClass();
        DerivedClass dc = new DerivedClass();

        // The following line causes compiler error CS0266.
        dc = mc;

        // You can resolve the error by using an explicit conversic
        dc = (DerivedClass)mc;
    }
}

class DerivedClass : MyClass
{
}
```

Compiler Error CS0269

使用了未赋值的 out 参数“parameter”

编译器未能验证在使用 out 参数之前是否已给它赋值，它的值在赋值时可能未定义。在访问值之前，请确保为调用用的方法中的 **out** 参数赋值。如果您需要使用传入的变量的值，请改用 **ref** 参数。有关更多信息，请参见传递参数（C# 编程指南）。

下面的示例生成 CS0269：

```
// CS0269.cs
class C
{
    public static void F(out int i)
    // One way to resolve the error is to use a ref parameter inste
    // of an out parameter.
    //public static void F(ref int i)
    {
        // The following line causes a compiler error because no va
        // has been assigned to i.
        int k = i;  // CS0269
        i = 1;
        // The error does not occur if the order of the two previou
        // lines is reversed.
    }

    public static void Main()
    {
        int myInt = 1;
        F(out myInt);
        // If the declaration of method F is changed to require a r
        // parameter, ref must be specified in the call as well.
        //F(ref myInt);
    }
}
```

如果变量的初始化发生在 try 块中，则也会出现此错误，因为编译器无法验证 try 块是否可以成功执行：

```
// CS0269b.cs
class C
{
    public static void F(out int i)
    {
        try
        {
            // Assignment occurs, but compiler can't verify it
            i = 1;
        }
        catch
        {
        }

        int k = i;  // CS0269
        i = 1;
    }

    public static void Main()
    {
        int myInt;
        F(out myInt);
    }
}
```

Compiler Error CS0270

不能在变量声明中指定数组大小(请尝试使用“new”表达式初始化)

将大小指定为数组声明的一部分时会发生此错误。若要解决此错误，请使用 new 运算符表达式。

下面的示例生成 CS0270：

```
// CS0270.cs
// compile with: /t:module

public class Test
{
   int[10] a;   // CS0270
   // To resolve, use the following line instead:
   // int[] a = new int[10];
}
```

Compiler Error CS0304

变量类型“type”没有 new() 约束，因此无法创建该类型的实例

如果实现泛型类，并且需要使用 **new** 关键字来创建为类型参数 T 提供的任何类型的新实例，则必须在类声明中将 new() 约束应用于 T，如下面的示例所示。

```
class C<T> where T : new()
```

new() 约束将通过确保为 T 提供的任何具体类型都具有默认值（无参数构造函数）来实施类型安全。如果您在 T 未指定 **new()** 约束时尝试在类的正文中使用 **new** 运算符创建类型参数 T 的实例，将出现 CS0304。在客户端，如果代码尝试使用没有默认构造函数的类型实例化泛型类，则该代码将生成 Compiler Error CS0310。

下面的示例会生成 CS0304。

```
// CS0304.cs
// Compile with: /target:library.
class C<T>
{
    // The following line generates CS0304.
    T t = new T();
}
```

该类的方法中也不允许使用 **new** 运算符。

```
// Compile with: /target:library.
class C<T>
{
    public void ExampleMethod()
    {
        // The following line generates CS0304.
        T t = new T();
    }
}
```

若要避免该错误，请使用 **new()** 约束声明类，如下面的示例所示。

```
// Compile with: /target:library.
class C<T> where T : new()
{
    T t = new T();

    public void ExampleMethod()
    {
        T t = new T();
    }
}
```

请参阅

C# Compiler Errors

Compiler Error CS0310

类型“typename”必须是具有公共的无参数构造函数的非抽象类型，才能用作泛型类型或方法“generic”中的参数“parameter”

泛型类型或方法在它的 where 子句中定义了一个新约束，要求任何类型均必须具有公共的无参数构造函数才能用作该泛型类型或方法的类型参数。若要避免此错误，请确保类型具有正确的构造函数，或者修改泛型类型或方法的约束子句。

下面的示例生成 CS0310：

```
// CS0310.cs
using System;

class G<T> where T : new()
{
    T t;

    public G()
    {
        t = new T();
        Console.WriteLine(t);
    }
}

class B
{
    private B() { }
    // Try this instead:
    // public B() { }
}

class CMain
{
    public static void Main()
    {
        G<B> g = new G<B>();   // CS0310
        Console.WriteLine(g.ToString());
    }
}
```

Compiler Error CS0311

不能将类型“type1”用作泛型类型或 '<name>方法中的类型参数“T”。没有从“type1”到“type2”的隐式引用转换。

在将某一约束应用于泛型类型参数时，必须存在从具体参数到该约束类型的隐式标识或引用转换。

更正此错误

1. 更改要在创建类时使用的参数。
2. 如果您拥有类，则可移除约束或执行某一操作以启用隐式引用或标识转换。例如，可以使第二个类型从第一个类型继承。

```
// cs0311.cs
class B{}
class C{}
class Test<T> where T : C
{ }

class Program
{
    static void Main()
    {
        Test<B> test = new Test<B>(); //CS0311
    }
}
```

如果在尝试使用值类型参数时发生此错误，您会发现隐式数值转换（例如，从 **short** 到 **int** 的转换）不满足泛型类型参数的约束。

请参阅

类型参数的约束（C# 编程指南）

Compiler Error CS0413

由于类型参数“type parameter”既没有类类型约束，也没有“class”约束，因此不能与“as”运算符一起使用

如果某个泛型类型使用了 as 运算符，但该泛型类型不具有类类型约束，则发生此错误。 **as** 运算符只允许用于引用类型，因此必须约束类型参数以保证它不是值类型。要避免此错误，请使用类类型约束或引用类型约束。

这是因为 **as** 运算符能够返回 **null**（它不可能是值类型的值），并且类型参数必须被视为值类型，除非它是类类型约束或引用类型约束。

下面的示例生成 CS0413。

```
// CS0413.cs
// compile with: /target:library
class A {}
class B : A {}

class CMain
{
   A a = null;
   public void G<T>()
   {
      a = new A();
      System.Console.WriteLine (a as T);  // CS0413
   }

   // OK
   public void H<T>() where T : A
   {
      a = new A();
      System.Console.WriteLine (a as T);
   }
}
```

Compiler Error CS0417

“identifier”：创建变量类型的实例时无法提供参数

如果对类型参数上的 **new**的调用带有参数，则会发生此错误。通过在未知参数类型上的使用**new**，可以被调用的唯一构造函数是不带参数的。如果需要调用另一个构造函数，请考虑使用类类型约束或接口约束。

下面的示例生成 CS0417：

```
// CS0417
class ExampleClass<T> where T : new()
{
    // The following line causes CS0417.
    T instance1 = new T(1);

    // The following line doesn't cause the error.
    T instance2 = new T();
}
```

请参阅

类型参数的约束（C# 编程指南）

Compiler Error CS0433

类型 TypeName1 同时存在于 TypeName2 和 TypeName3 中。

在应用程序中引用的两个不同的程序集包含相同的命名空间和类型，这会产生混乱。

若要解决此错误，请使用 /reference (C# Compiler Options) 编译器选项的别名功能，或者不引用您的程序集。

此代码用歧义类型的第一个副本创建 DLL。

```
// CS0433_1.cs
// compile with: /target:library
namespace TypeBindConflicts
{
   public class AggPubImpAggPubImp {}
}
```

此代码用歧义类型的第二个副本创建 DLL。

```
// CS0433_2.cs
// compile with: /target:library
namespace TypeBindConflicts
{
   public class AggPubImpAggPubImp {}
}
```

下面的示例生成 CS0433。

```
// CS0433_3.cs
// compile with: /reference:cs0433_1.dll /reference:cs0433_2.dll
using TypeBindConflicts;
public class Test
{
   public static void Main()
   {
      AggPubImpAggPubImp n6 = new AggPubImpAggPubImp();   // CS043
   }
}
```

下面的示例演示如何使用 **/reference** 编译器选项的别名功能来解决此 CS0433 错误。

```
// CS0433_4.cs
// compile with: /reference:cs0433_1.dll /reference:TypeBindConflic
using TypeBindConflicts;
public class Test
{
   public static void Main()
   {
      AggPubImpAggPubImp n6 = new AggPubImpAggPubImp();
   }
}
```

Compiler Error CS0445

无法修改取消装箱转换的结果

取消装箱转换的结果是一个临时变量。编译器禁止您修改这样的变量，因为当临时变量消失时，任何修改也随之消失。若要修复此错误，声明新的值类型变量来存储中间表达式，并将取消装箱的转换的结果赋给该新变量。

下面的代码生成 CS0455。

```
// CS0445.CS
class UnboxingTest
{
    public static void Main()
    {
        Point p;
        p.x = 1;
        p.y = 2;
        object obj = p;
        // The following line generates CS0445, because the result
        // of unboxing obj is a temporary variable.
        ((Point)obj).x = 2;

        // The following lines resolve the error.

        // Store the result of the unboxing conversion in p2.
        Point p2;
        p2 = (Point)obj;
        // Then you can modify the unboxed value.
        p2.x = 2;
    }
}

struct Point
{
    public int x, y;
}
```

Compiler Error CS0446

Foreach 不能在“方法或委托”上运行。是否希望调用“方法或委托”？

指定一个没有括号的方法或指定一个在 **foreach** 语句部分（通常放入集合类）没有括号的匿名方法会导致此错误。请注意，如果方法返回一个集合类，则将该方法调用放在那个位置是有效的，尽管这不常见。

下面的代码将生成 CS0446。

```
// CS0446.cs
using System;
class Tester
{
    static void Main()
    {
        int[] intArray = new int[5];
        foreach (int i in M) { } // CS0446
    }
    static void M() { }
}
```

Compiler Error CS0504

常数“variable”无法标记为 static

如果变量是 const，它也就是 static。如果需要一个既是 **const** 又是 **static** 的变量，只需将该变量声明为 **const**；如果需要的只是 **static** 变量，只需将其标记为 **static**。

下面的示例生成 CS0504：

```
// CS0504.cs
namespace x
{
   abstract public class clx
   {
      static const int i = 0;   // CS0504, cannot be both static ar
      abstract public void f();
   }
}
```

编译器错误 CS0507

“function1”：当重写“access”继承成员“function2”时，无法更改访问修饰符

尝试在方法重写中更改访问规范。

示例

下面的示例生成 CS0507。

```
// CS0507.cs
abstract public class clx
{
   virtual protected void f() {}
}

public class cly : clx
{
   public override void f() {}   // CS0507
   public static void Main() {}
}
```

当类尝试重写在引用的元数据中定义的标记为 protected internal 的方法时，也会发生 CS0507。 在这种情况下，重写方法应标记为 protected。

```
// CS0507_b.cs
// compile with: /target:library
abstract public class clx
{
   virtual protected internal void f() {}
}
```

下面的示例生成 CS0507。

```
// CS0507_c.cs
// compile with: /reference:cs0507_b.dll
public class cly : clx
{
   protected internal override void f() {}   // CS0507
   // try the following line instead
   // protected override void f() {}   // OK

   public static void Main() {}
}
```

Compiler Error CS0518

预定义的类型“type”未定义或未导入

引起该问题的主要原因是项目没有导入 mscorlib.dll，该文件定义整个 System 命名空间。这可能由以下原因之一导致：

- 指定了命令行编译器的 /nostdlib 选项。/nostdlib 选项将禁止导入 mscorlib.dll。如果想要定义或创建用户特定的 System 命名空间，请使用此选项。
- 引用了错误的 mscorlib.dll。
- Visual Studio .NET 或 .NET Framework 公共语言运行时安装存在损坏。
- 早期安装所剩的组件与最新安装的其余组件不兼容。

要解决此问题，请采用以下操作之一：

- 不指定命令行编译器的 /nostdlib 选项。
- 确保项目引用正确的 mscorlib.dll。
- 重新安装 .NET Framework 公共语言运行时（若以上解决方案均未能解决此问题）。

Compiler Error CS0523

“struct1”类型的结构成员“struct2 field”在结构布局中导致循环

两个结构的定义包含递归引用。更改 struct 定义以便每个结构都不会在另一结构上定义自身。此限制仅适用于结构，因为结构是值类型。如果使用递归引用，请将类型声明为类。

下面的示例生成 CS0523：

```
// CS0523.cs
// compile with: /target:library
struct RecursiveLayoutStruct1
{
   public RecursiveLayoutStruct2 field;
}

struct RecursiveLayoutStruct2
{
   public RecursiveLayoutStruct1 field;   // CS0523
}
```

Compiler Error CS0545

“function”：无法重写，因为“property”没有可重写的 get 访问器

尝试定义属性访问器的重写，而此时基类没有这样的定义可供重写。可以通过下面的方法解决该错误：

- 在基类中添加 **set** 访问器。
- 从派生类中移除 **set** 访问器。
- 通过在派生类的属性中添加 new 关键字来隐藏基类属性。
- 生成基类属性 virtual。

有关更多信息，请参见使用属性（C# 编程指南）。

下面的示例生成 CS0545。

```
// CS0545.cs
// compile with: /target:library
// CS0545
public class a
{
   public virtual int i
   {
      set {}

      // Uncomment the following line to resolve.
      // get { return 0; }
   }
}

public class b : a
{
   public override int i
   {
      get { return 0; }
      set {}   // OK
   }
}
```

Compiler Error CS0552

“conversion routine”：与接口之间的用户定义转换

不能创建用户定义的接口转换。如果需要转换例程，则使接口成为类来解决该错误，或者从接口派生类。

下面的示例生成 CS0552：

```
// CS0552.cs
public interface ii
{
}

public class a
{
   // delete the routine to resolve CS0552
   public static implicit operator ii(a aa) // CS0552
   {
      return new ii();
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0563

二元运算符的参数之一必须是包含类型

运算符重载的方法声明必须遵循一定的准则。

下面的示例生成 CS0563：

```
// CS0563.cs
public class iii
{
    public static implicit operator int(iii x)
    {
        return 0;
    }
    public static implicit operator iii(int x)
    {
        return null;
    }
    public static int operator +(int aa, int bb)   // CS0563
    // Use the following line instead:
    // public static int operator +(int aa, iii bb)
    {
        return 0;
    }
    public static void Main()
    {
    }
}
```

Compiler Error CS0570

属性、索引器或事件“name”不受该语言支持；请尝试直接调用访问器方法“name!”

当使用由另一编译器生成的导入元数据时发生此错误。您的代码尝试使用编译器无法处理的类成员。

下面的 C++ 程序使用了一种其他语言不能使用的特性 RequiredAttributeAttribute。

```
// CPP0570.cpp
// compile with: /clr /LD

using namespace System;
using namespace System::Runtime::CompilerServices;

namespace CS0570_Server {
   [RequiredAttributeAttribute(Int32::typeid)]
   public ref struct Scenario1 {
      int intVar;
   };

   public ref struct CS0570Class {
      Scenario1 ^ sc1_field;

      property virtual Scenario1 ^ sc1_prop {
         Scenario1 ^ get() { return sc1_field; }
      }

      Scenario1 ^ sc1_method() { return sc1_field; }
   };
};
```

下面的示例生成 CS0570。

```
// CS0570.cs
// compile with: /reference:CPP0570.dll
using System;
using CS0570_Server;

public class C {
   public static int Main() {
      CS0570Class r = new CS0570Class();
      r.sc1_field = null;   // CS0570
      object o = r.sc1_prop;   // CS0570
      r.sc1_method();   // CS0570
   }
}
```

Compiler Error CS0571

“function”：无法显式调用运算符或访问器

某些运算符具有内部名称。例如，**op_Increment** 是 ++ 运算符的内部名称。不应使用或显式调用这样的方法名称。

下面的示例生成 CS0571：

```
// CS0571.cs
public class MyClass
{
   public static MyClass operator ++ (MyClass c)
   {
      return null;
   }

   public static int prop
   {
      get
      {
         return 1;
      }
      set
      {
      }
   }

   public static void Main()
   {
      op_Increment(null);   // CS0571
      // use the increment operator as follows
      // MyClass x = new MyClass();
      // x++;

      set_prop(1);      // CS0571
      // try the following line instead
      // prop = 1;
   }
}
```

Compiler Error CS0579

重复“attribute”特性

不可能多次指定相同的特性，除非特性在其 AttributeUsage 中指定 **AllowMultiple=true**。

下面的示例会产生 CS0579。

```
// CS0579.cs
using System;
public class MyAttribute : Attribute
{
}

[AttributeUsage(AttributeTargets.All,AllowMultiple=true)]
public class MyAttribute2 : Attribute
{
}

public class z
{
    [MyAttribute, MyAttribute]     // CS0579
    public void zz()
    {
    }

    [MyAttribute2, MyAttribute2]   // OK
    public void zzz()
    {
    }

    public static void Main()
    {
    }
}
```

Compiler Error CS0592

特性“attribute”对此声明类型无效。它只对“type”声明有效。

在定义特性时，可通过指定 AttributeTargets 值来定义可将该特性应用于什么构造。在下面的示例中，MyAttribute 特性只能应用于接口，因为 AttributeUsage 特性指定 AttributeTargets.Iterface。之所以会生成错误，原因是该特性应用于类（类 A）。

下面的示例生成 CS0592：

```
// CS0592.cs
using System;

[AttributeUsage(AttributeTargets.Interface)]
public class MyAttribute : Attribute
{
}

[MyAttribute]
// Generates CS0592 because MyAttribute is not valid for a class.
public class A
{
    public static void Main()
    {
    }
}
```

请参阅

特性（C# 和 Visual Basic）

Compiler Error CS0616

“class”不是特性类

尝试在特性块中使用非特性类。所有特性类型都必须从 System.Attribute 继承。

下面的示例生成 CS0616。

```
// CS0616.cs
// compile with: /target:library
[CMyClass(i = 5)]   // CS0616
public class CMyClass {}
```

下面的示例显示您可以如何定义特性：

```
// CreateAttrib.cs
// compile with: /target:library
using System;

[AttributeUsage(AttributeTargets.Class|AttributeTargets.Interface)]
public class MyAttr : Attribute
{
   public int Name = 0;
   public int Count = 0;

   public MyAttr (int iCount, int sName)
   {
      Count = iCount;
      Name = sName;
   }
}

[MyAttr(5, 50)]
class Class1 {}

[MyAttr(6, 60)]
interface Interface1 {}
```

Compiler Error CS0650

错误的数组声明符：若要声明托管数组，秩说明符应位于变量的标识符之前。若要声明固定大小缓冲区字段，应在字段类型之前使用 fixed 关键字。

数组未正确声明。在 C# 中，与 C 和 C++ 中不同，方括号放在类型（而不是变量名）后面。并且，请注意固定大小缓冲区的语法与数组的语法不同。

下面的代码示例生成 CS0650。

```
// CS0650.cs
public class MyClass
{
   public static void Main()
   {
// Generates CS0650\. Incorrect array declaration syntax:
      int myarray[2];

      // Correct declaration.
      int[] myarray2;

      // Declaration and initialization in one statement
      int[] myArray3= new int[2] {1,2}

      // Access an array element.
      myarray3[0] = 0;
    }
}
```

下面的示例演示如何在创建固定大小缓冲区时使用 **fixed** 关键字：

```
// This code must appear in an unsafe block.
public struct MyArray
{
    public fixed char pathName[128];
}
```

请参阅

固定大小的缓冲区（C# 编程指南）

fixed 语句（C# 参考）

数组（C# 编程指南）

Compiler Error CS0686

访问符“accessor”无法实现类型“type”的接口成员“member”。使用显式接口实现。

提示：当实现一个接口时，如果其中包含的方法名称与属性或事件关联的自动生成的方法相冲突，会出现此错误。属性的 get/set 方法生成为 get_property 和 set_property，事件的 add/remove 方法生成为 add_event 和 remove_event。如果接口包含其中的任何一个方法，则发生冲突。要避免此错误，请使用显式接口实现来实现这些方法。要这样做，请将函数指定为：

```
      Interface.get_property() { /* */ }
Interface.set_property() { /* */ }
```

以下示例生成 CS0686：

```
// CS0686.cs
interface I
{
    int get_P();
}

class C : I
{
    public int P
    {
        get { return 1; }  // CS0686
    }
}
// But the following is valid:
class D : I
{
    int I.get_P() { return 1; }
    public static void Main() {}
}
```

声明事件时也会发生此错误。事件结构自动生成 add***event*** 和 *remove***event** 方法，这可能和接口中的同名方法相冲突，如下示例所示：

```
// CS0686b.cs
using System;

interface I
{
    void add_OnMyEvent(EventHandler e);
}

class C : I
{
    public event EventHandler OnMyEvent
    {
        add { }  // CS0686
        remove { }
    }
}

// Correct (using explicit interface implementation):
class D : I
{
    void I.add_OnMyEvent(EventHandler e) {}
    public static void Main() {}
}
```

Compiler Error CS0702

约束不能是特殊类“identifier”

下面的类型不能用作约束：System.Object、System.Array、System.Delegate、System.Enum 和 System.ValueType。

下面的示例生成 CS0702：

```
// CS0702.cs
class C<T> where T : System.Array  // CS0702
{
}
```

请参阅

类型参数的约束（C# 编程指南）

编译器错误 CS0703

可访问性不一致：约束类型“identifier”的可访问性比“identifier”低。

一个约束不能强制泛型参数比泛型类本身的可访问性低。 在下面的示例中，虽然泛型类 CC<T>被声明为 public，但约束尝试强制 T 实现内部接口。 即使允许这样做，也只有具有内部访问的客户端才能为该类创建参数，因此实际上只有具有内部访问的客户端才能使用该类。

为消除此错误，请确保泛型类的访问级别的限制性不低于界限中出现的任何类或接口。

下面的示例生成 CS0703：

```
// CS0703.cs
internal interface I {}
public class C<T> where T : I  // CS0703 – I is internal; C<T> is p
{
}
```

Compiler Error CS0731

程序集“assembly”中类型“type”的类型转发器导致循环

仅当导入的源数据格式错误时才会发生此错误。仅使用 C# 源代码不会出现此错误。

下面的示例生成 CS0731。此示例由三个文件组成：

1.Circular.IL

2.Circular2.IL

3.CS0731.cs

首先将 .IL 文件编译为库，然后编译引用这两个文件的 .cs 代码。

```
// Circular.il
// compile with: /DLL /out=Circular.dll
.assembly extern circular2
{
  .ver 0:0:0:0
}
.assembly extern circular3
{
  .ver 0:0:0:0
}
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )   // .z\V.4..
  .ver 2:0:0:0
}
.assembly Circular
{
  .custom instance void [mscorlib]System.Runtime.CompilerServices.C
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.class extern forwarder Circular.Referenced.TypeForwarder
{
  .assembly extern circular2
}
.module Circular.dll
// MVID: {880C2329-C915-42A0-83E9-9D10C3E6DBD0}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x04E40000
// ======== CLASS MEMBERS DECLARATION =========
.class public abstract auto ansi sealed beforefieldinit User
       extends [mscorlib]System.Object
{
  .method public hidebysig static class [circular2]Circular.Referer
          F() cil managed
  {
    .maxstack  1
    newobj     instance void [circular2]Circular.Referenced.TypeFor
    ret
  }
}
```

```
// Circular2.il
// compile with: /DLL /out=Circular2.dll
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )   // .z\V.4..
  .ver 2:0:0:0
}
.assembly extern Circular
{
  .ver 0:0:0:0
}
.assembly circular2
{
  .custom instance void [mscorlib]System.Runtime.CompilerServices.C
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.class extern forwarder Circular.Referenced.TypeForwarder
{
  .assembly extern Circular
}
.module circular2.dll
// MVID: {8B3BE5C8-DBE1-49C4-BC72-DF35F0387C21}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       //   WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x04E40000
```

```
// CS0731.cs
// compile with: /reference:circular.dll /reference:circular2.dll
// CS0731 expected
class A {
   public static void Main() {
      User.F();
   }
}
```

Compiler Error CS0826

找不到隐式类型数组的最佳类型。

数组元素都必须为同一类型，或者可根据编译器所使用的类型推理规则隐式转换为同一类型。最佳类型必须为数组表达式中的类型之一。元素不会转换为诸如 **object** 之类的新类型。对于隐式类型的数组，编译器必须根据为其分配的元素类型推断该数组的类型。

更正此错误

- 为该数组指定显式类型。
- 为所有数组元素指定同一类型。
- 为可能会导致问题的那些元素提供显式强制转换。

下面的代码生成 CS0826，因为数组元素的类型不完全相同，并且编译器的类型推理逻辑未找到一个最佳类型：

```
// cs0826.cs
public class C
{
    public static int Main()
    {
        var x = new[] { 1, "str" }; // CS0826

        char c = 'c';
        short s1 = 0;
        short s2 = -0;
        short s3 = 1;
        short s4 = -1;

        var array1 = new[] { s1, s2, s3, s4, c, '1' }; // CS0826
        return 1;
    }
}
```

请参阅

隐式类型的局部变量（C# 编程指南）

Compiler Error CS0834

lambda 表达式必须具有表达式体才能转换为表达式树。

要转换为表达式树的 lambda 必须为表达式 lambda；语句 lambda 和匿名方法只能转换为委托类型。

更正此错误

1. 从 lambda 表达式中移除语句。

下面的示例生成 CS0834：

```
// cs0834.cs
using System;
using System.Linq;
using System.Linq.Expressions;

public class C
{
    public static int Main()
    {
        Expression<Func<int, int>> e = x => { return x; }; // CS083
    }
}
```

Compiler Error CS0840

“Property name”必须声明主体，因为它未标记为 abstract 或 extern。自动实现的属性必须同时定义 get 和 set 访问器。

除非常规属性被标记为 **abstract** 或 **extern** 或者作为 **partial** 类型的成员，否则必须为其提供主体。自动实现的属性不提供访问器体，但它们必须指定两个访问器。若要创建只读的自动实现属性，请将 set 访问器设为 **private**。

更正此错误

1. 提供缺少的主体或访问器，或者在该属性和/或其封闭类型上使用 abstract、extern 或 分部（类型）（C# 参考） 修饰符。

下面的示例生成 CS0840：

```
// cs0840.cs
// Compile with /target:library
using System;
class Test
{
    public int myProp { get; } // CS0840

    // to create a read-only property
    // try the following line instead
    public int myProp2 { get; private set; }

}
```

请参阅

自动实现的属性（C# 编程指南）

编译器错误 CS0843

在控制返回到调用程序之前，自动实现的属性“name”的支持字段必须完全赋值。请考虑从构造函数初始值设定项中调用默认构造函数。

若要从构造函数中为自动实现的属性赋值，必须首先调用默认构造函数以创建对象。

更正此错误

- 在构造函数初始值设定项中添加对默认构造函数的调用，如以下示例中所示。请注意：this() 的使用。 有关更多信息，请参见 this（C# 参考）。

示例

下面的代码将生成 CS0843：

```
// cs0843.cs
struct S
{
    public int AIProp { get; set; }
    public S(int i){} //CS0843
    // Try the following lines instead.
   // public S(int i) : this()
   // {
   //     AIProp = i;
   // }
}

class Test
{
    static int Main()
    {
        return 1;
    }
}
```

Compiler Error CS0845

表达式树 lambda 不能在合并运算符左侧包含 null 文本。

由于 null 本身不具有类型，因此 null 合并运算符无法对其进行运算。

更正此错误

1. 将 null 文本强制转换为对象。

下面的代码将生成 CS0845：

```
// cs0845.cs
using System;
using System.Linq;
using System.Linq.Expressions;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Expression<Func<object>> e = () => null ?? null; // CS(
            // Try the following line instead.
            // Expression<Func<object>> e = () => (object)null ?? r
        }
    }
}
```

Compiler Error CS1001

应输入标识符

未提供标识符。标识符是您提供的类、结构、命名空间、方法、变量等的名称。

下面的示例声明一个简单的类，但没有为类指定名称：

```
//cs1001.cs
public class              //CS1001
    {
        public int Num {get; set;}
        void MethodA(){}
    }
```

下面的示例之所以会生成 CS1001，原因是在声明枚举时必须指定成员：

```
// CS1001.cs
public class clx
{
   enum Colors : int
   {
      'a', 'b' // CS1001, 'a' is not a valid int identifier
       // The following line shows examples of valid identifiers:
       // Blue, Red, Orange
   };

   public static void Main()
   {
   }
}
```

即使编译器不使用参数名（例如，在接口定义中），也需要参数名。需要这些参数，以便使用接口的程序员能够对参数的含义有所了解。

```
// CS1001-2.cs
// compile with: /target:library
interface IMyTest
{
   void TestFunc1(int, int);  // CS1001
   // Use the following line instead:
   // void TestFunc1(int a, int b);
}

class CMyTest : IMyTest
{
   void IMyTest.TestFunc1(int a, int b)
   {
   }
}
```

请参阅

语句、表达式和运算符（C# 编程指南）

类型（C# 编程指南）

Compiler Error CS1009

无法识别的转义序列

在 z字符串 中反斜杠 () 的后面是一个意外的字符。编译器需要一个有效的转义字符。有关详细信息，请参阅字符 转义。

下面的示例生成 CS1009。

```
// CS1009-a.cs
class MyClass
{
   static void Main()
   {
      // The following line causes CS1009.
      string a = "\m";
      // Try the following line instead.
      // string a = "\t";
   }
}
```

发生该错误的原因通常是在文件名中使用了反斜杠字符，如下面的示例所示：

```
string filename = "c:\myFolder\myFile.txt";
```

若要纠正该错误，请使用"\\"或前面带有 @ 且用引号括起的字符串，如下面的示例所示：

```
// CS1009-b.cs
class MyClass
{
   static void Main()
   {
      // The following line causes CS1009.
      string filename = "c:\myFolder\myFile.txt";
      // Try one of the following lines instead.
      // string filename = "c:\\myFolder\\myFile.txt";
      // string filename = @"c:\myFolder\myFile.txt";
   }
}
```

请参阅

string（C# 参考）

Compiler Error CS1018

应输入关键字“this”或“base”

编译器遇到了不完整的构造函数声明。

下面的示例生成 CS1018 并建议了几种解决此错误的方法：

```
// CS1018.cs
public class C
{
}

public class a : C
{
    public a(int i)
    {
    }

    public a () :   // CS1018
    // possible resolutions:
    // public a () resolves by removing the colon
    // public a () : base() calls C's default constructor
    // public a () : this(1) calls the assignment constructor of cl
    {
    }

    public static int Main()
    {
        return 1;
    }
}
```

Compiler Error CS1019

应输入可重载的一元运算符

声明了类似于重载的一元运算符的内容，但缺少该运算符，或该运算符在签名中的位置不正确。

“一元运算符”是一种对单个操作数进行运算的运算符。例如，**++** 是一元运算符。您可以通过使用 **operator** 关键字并指定运算符所运算的类型的单一参数来重载某些一元运算符。例如，如果要为用户定义的类 Temperature 重载运算符 **++** 以便能够写入 Temperature++，您可以采用以下方式定义该类：

```
public static  Temperature operator ++ (Temperature temp)
{
    temp.Degrees++;
    return temp;
}
```

如果收到此错误，则表明您声明了类似于重载的一元运算符的内容，但缺少该运算符本身，或其在签名中的位置不正确。如果在前面的示例中从签名中移除了 ++，将会生成 CS1019。

下面的代码生成 CS1019：

```
// CS1019.cs
public class ii
{
   int i
   {
      get
      {
         return 0;
      }
   }
}

public class a
{
    public int i;
// Generates CS1019: "ii" is not a unary operator.
   public static a operator ii(a aa)

   // Use the following line instead:
   //public static a operator ++(a aa)
   {
      aa.i++;
      return aa;
   }

   public static void Main()
   {
   }
}
```

请参阅

运算符（C# 编程指南）

Compiler Error CS1026

应输入)

找到了不完整的语句。

发生该错误的通常原因是将语句而不是表达式放入 ASP.NET 页中的内联表达式内。例如，以下表达式是错误的：

```
<%=new TimeSpan(DateTime.Now.Ticks - new DateTime(2001, 1, 1).Ticks
```

正确的表达式应如下：

```
<%=new TimeSpan(DateTime.Now.Ticks - new DateTime(2001, 1, 1).Ticks
```

其解释如下：

```
<% Response.Write(new TimeSpan(DateTime.Now.Ticks - new DateTime(20
```

下面的示例生成 CS1026：

```
// CS1026.cs
#if (a == b   // CS1026, add closing )
#endif

class x
{
   public static void Main()
   {
   }
}
```

Compiler Error CS1029

error：“text”

显示用 #error 指令定义的错误文本。

下面的示例显示如何创建用户定义的错误：

```
// CS1029.cs
class Sample
{
   static void Main()
   {
      #error Let's give an error here   // CS1029
   }
}
```

Compiler Error CS1061

“type”不包含“member”的定义，并且找不到可接受类型为“type”的第一个参数的扩展方法“name”(是否缺少 using 指令或程序集引用?)。

在尝试调用的方法或访问的类成员不存在时，将发生此错误。

下面的示例生成 CS1061，因为 TestClass1 没有 DisplaySomething 方法。它没有名为 WriteSomething 的方法。该方法可能是此源代码的作者打算编写的方法。

```
// cs1061.cs
public class TestClass1
{
    // TestClass1 has one method, called WriteSomething.
    public void WriteSomething(string s)
    {
        System.Console.WriteLine(s);
    }
}

public class TestClass2
{
    // TestClass2 has one method, called DisplaySomething.
    public void DisplaySomething(string s)
    {
        System.Console.WriteLine(s);
    }
}

public class TestTheClasses
{
    public static void Main()
    {
        TestClass1 tc1 = new TestClass1();
        TestClass2 tc2 = new TestClass2();
        // The following call fails because TestClass1 does not hav
        // a method called DisplaySomething.
        tc1.DisplaySomething("Hello");      // CS1061

        // To correct the error, change the method call to either
        // tc1.WriteSomething or tc2.DisplaySomething.
        tc1.WriteSomething("Hello from TestClass1");
        tc2.DisplaySomething("Hello from TestClass2");
    }
}
```

请参阅

类和结构（C# 编程指南）

扩展方法（C# 编程指南）

Compiler Error CS1112

不要使用“System.Runtime.CompilerServices.ExtensionAttribute”。请改用“this”关键字。

当在包含扩展方法的非静态类上使用 ExtensionAttribute 时，会生成此错误。如果在静态类上使用此特性，则可能会发生其他错误，如 CS0708：“不能在静态类中声明实例成员”。

在 C# 中，必须在静态类中定义扩展方法，并用 **this** 关键字修饰该方法的第一个参数。任何情况下都不要在源代码中使用该特性。有关更多信息，请参见扩展方法（C# 编程指南）。

更正此错误

1. 移除该特性，然后将 **this** 修饰符应用于该方法的第一个参数。

下面的示例生成 CS1112：

```
// cs1112.cs
[System.Runtime.CompilerServices.ExtensionAttribute] // CS1112
public class Extensions
{
    public bool A(bool b) { return b; }
}

class A { }
```

编译器错误 CS1501

“method”方法的重载不带有“number”个参数

对类方法进行了调用，但没有带有所需数目的参数的方法的定义。

下面的示例生成 CS1501。

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            ec.ExampleMethod();
            ec.ExampleMethod(10);
            // The following line causes compiler error CS1501 beca
            // ExampleClass does not contain an ExampleMethod that
            // two arguments.
            ec.ExampleMethod(10, 20);
        }
    }

    // ExampleClass contains two overloads for ExampleMethod. One o
    // has no parameters and one has a single parameter.
    class ExampleClass
    {
        public void ExampleMethod()
        {
            Console.WriteLine("Zero parameters");
        }

        public void ExampleMethod(int i)
        {
            Console.WriteLine("One integer parameter.");
        }

        //// To fix the error, you must add a method that takes two
        //public void ExampleMethod (int i, int j)
        //{
        //    Console.WriteLine("Two integer parameters.");
        //}
    }
}
```

Compiler Error CS1502

与“name”最匹配的重载方法有一些无效参数

当正传递到方法的参数类型与该方法的参数类型不匹配时发生该错误。如果重载调用的方法，则重载的任何版本都不具有与正进行传递的参数类型相匹配的签名。

若要解决该问题，请执行以下操作：

- 仔细检查正进行传递的参数的类型。确保它们与正调用的方法的参数类型相匹配。
- 如果可以，请使用 Convert 类转换任何不匹配的参数。
- 如果可以，请强制转换所有不匹配的参数使它们与该方法需要的类型相匹配。
- 如果可以，请定义该方法的其他重载版本以便与正要发送的参数类型相匹配。

下面的示例生成 CS1502：

```
// CS1502.cs
namespace x
{
   public class a
   {
      public a(char i)
      // try the following constructor instead
      // public a(int i)
      {
      }

      public static void Main()
      {
         a aa = new a(2222);   // CS1502
      }
   }
}
```

Compiler Error CS1519

类、结构或接口成员声明中的标记“token”无效

每当在某个标记未隶属的位置中遇到该标记时，都会产生此错误。“标记”是关键字；标识符（类、结构、方法等的名称）；字符串、字符或数字文本值（例如 108、"Hello" 或 'A'）；或者是运算符或标点的形式（例如 == 或 ;）。

任何在类型的前面包含无效修饰符的 class、结构或接口的成员声明都将生成该错误。若要修复此错误，请移除无效的修饰符。

下面的示例会在五个位置生成 CS1519，因为标记放在其无效的位置：

```
// CS1519.cs
// Generates CS1519 because a class name cannot be a number:
class Test 42
{
// Generates CS1519 because of 'j' following 'I'
// with no comma between them:
    int i j;
// Generates CS1519 because of "checked" on void method:
    checked void f4();

// Generates CS1519 because of "num":
    void f5(int a num){}

// Generates CS1519 because of namespace inside class:
    namespace;

}
```

请参阅

类（C# 编程指南）

结构（C# 编程指南）

接口（C# 编程指南）

方法（C# 编程指南）

Compiler Error CS1540

无法通过类型“type1”的限定符访问保护成员“member”；限定符必须是类型“type2”(或者从该类型派生的)

派生的类无法通过基类的实例来访问其基类的受保护成员。在派生的类中声明的基类的实例在运行时可能是另一个类型的实例，该类型从相同的基派生但与派生的类无关。由于受保护成员只可由派生的类型访问，因此要访问可能在运行时无效的受保护成员的任何尝试都会由编译器标记为无效。

在以下示例中的 Employee 类中，emp2 和 emp3 在编译时都具有类型 Person，但 emp2 在运行时具有类型 Manager。由于 Employee 未从 Manager 派生，因此它不能通过 Manager 类的实例来访问基类 Person 的受保护成员。编译器无法确定两个 Person 对象的运行时类型将是什么类型。因此，来自 emp2 的调用和来自 emp3 的调用都会导致编译器错误 CS1540。

```
using System;
using System.Text;

namespace CS1540
{
    class Program1
    {
        static void Main()
        {
            Employee.PreparePayroll();
        }
    }

    class Person
    {
        protected virtual void CalculatePay()
        {
            Console.WriteLine("CalculatePay in Person class.");
        }
    }

    class Manager : Person
    {
        protected override void CalculatePay()
        {
            Console.WriteLine("CalculatePay in Manager class.");

        }
    }

    class Employee : Person
    {
        public static void PreparePayroll()
        {
            Employee emp1 = new Employee();
            Person emp2 = new Manager();
            Person emp3 = new Employee();
            // The following line calls the method in the Employee
            // Person.
            emp1.CalculatePay();

            // The following lines cause compiler error CS1540\. Th
            // cannot determine at compile time what the run-time t
            // emp2 and emp3 will be.
            //emp2.CalculatePay();
            //emp3.CalculatePay();

        }
    }
}
```

请参阅

继承（C# 编程指南）

多态性（C# 编程指南）

访问修饰符（C# 编程指南）

抽象类、密封类及类成员（C# 编程指南）

Compiler Error CS1546

属性、索引器或事件“property”不受该语言支持；请尝试直接调用访问器方法“accessor”

代码正在使用具有默认索引属性的对象，并尝试使用索引语法。若要解决该错误，请调用属性的访问器方法。有关索引器和属性的更多信息，请参见索引器（C# 编程指南）。

下面的示例生成 CS1546。

此代码示例由一个编译为 .dll 的 .cpp 文件和一个使用该 .dll 的 .cs 文件组成。下面是 .dll 文件的代码，它定义了一个将由 .cs 文件中的代码访问的属性。

```
// CPP1546.cpp
// compile with: /clr /LD
using namespace System;
public ref class MCPP
{
public:
    property int Prop [int,int]
    {
        int get( int i, int b )
        {
            return i;
        }
    }
};
```

这是 C# 文件。

```
// CS1546.cs
// compile with: /r:CPP1546.dll
using System;
public class Test
{
    public static void Main()
    {
        int i = 0;
        MCPP mcpp = new MCPP();
        i = mcpp.Prop(1,1); // CS1546
        // Try the following line instead:
        // i = mcpp.get_Prop(1,1);
    }
}
```

Compiler Error CS1548

对程序集“assembly”签名时加密失败 —“reason”

程序集签名失败时将发生 CS1548。这通常是由于无效的密钥文件名称、无效的密钥文件路径或已损坏的密钥文件而造成的。

若要对程序集进行完全签名，必须提供包含公钥和私钥信息的有效密钥文件。若要延迟对程序集的签名，必须选择“仅延迟签名”复选框，并提供包含公钥信息的有效密钥文件。当程序集为延迟签名时，不需要使用私有密钥。

有关更多信息，请参见 How to: Sign an Assembly (Visual Studio)、/keyfile (C# Compiler Options) 和 /delaysign (C# Compiler Options)。

创建程序集时，C# 编译器会调入一个名为“al.exe”的实用工具。如果在创建程序集时出现失败，al.exe 会报告失败的原因。请参见 Al.exe Tool Errors and Warnings，并搜索与编译器在“原因”中报告的文本相对应的主题。

请参阅

How to: Sign an Assembly (Visual Studio)

Compiler Error CS1564

指定的选项冲突: Win32 资源文件；Win32 清单。

如果使用 **/Win32res** 编译器选项，则必须在资源文件中包含自定义 Win32 清单（如果需要）。自定义 Win32 清单必须与 Win32 资源文件一起提供。只有在未指定 win32 资源文件时，才使用 **/win32manifest** 选项。

更正此错误

1. 将 win32 清单添加到 win32 资源文件中，并移除 **/win32manifest** 编译器选项。

下面的示例在用 **/Win32res** 选项编译并且资源文件中未包含清单时会生成 CS1564。

```
// cs1564.cs
// Compile with: /Win32res
public class Test
{
    static int Main(string[] args)
    {
        return 1;
    }
}
```

C# 3.0 编译器为所有二进制文件添加了默认的 win32Manifest。

请参阅

/win32manifest (C# Compiler Options)

/win32res (C# Compiler Options)

Compiler Error CS1567

生成 Win32 资源时出错：“file”

您的编译或者使用了 /win32icon 编译器选项，或者未使用 /win32res 编译器选项，这导致编译器生成包含资源信息的文件，但编译器因磁盘空间不足或其他某些错误而无法创建文件。

如果您无法解决此文件生成问题，可以使用 /win32res，该编译器选项不生成包含资源信息的文件。

Compiler Error CS1579

foreach 语句不能对“type1”类型的变量进行操作，因为“type2”不包含“identifier”的公共定义

若要使用 foreach 语句循环访问某个集合，该集合必须满足以下要求：

- 它必须是一个接口、类或结构。
- 它必须包含一个返回类型的公共 GetEnumerator 方法。
- 返回类型必须包含一个名为 Current 的公共属性和一个名为 MoveNext 的公共方法。
- 有关更多信息，请参见 如何：使用 foreach 访问集合类（C# 编程指南）。

在本示例中，foreach 无法循环访问集合，因为 MyCollection 集合中没有 **public**GetEnumerator 方法。

下面的示例生成 CS1579。

```
// CS1579.cs
using System;
public class MyCollection
{
   int[] items;
   public MyCollection()
   {
      items = new int[5] {12, 44, 33, 2, 50};
   }

   // Delete the following line to resolve.
   MyEnumerator GetEnumerator()

   // Uncomment the following line to resolve:
   // public MyEnumerator GetEnumerator()
   {
      return new MyEnumerator(this);
   }

   // Declare the enumerator class:
   public class MyEnumerator
   {
      int nIndex;
      MyCollection collection;
      public MyEnumerator(MyCollection coll)
      {
         collection = coll;
         nIndex = -1;
      }
```

```
      public bool MoveNext()
      {
         nIndex++;
         return(nIndex < collection.items.GetLength(0));
      }

      public int Current
      {
         get
         {
            return(collection.items[nIndex]);
         }
      }
   }

   public static void Main()
   {
      MyCollection col = new MyCollection();
      Console.WriteLine("Values in the collection are:");
      foreach (int i in col)   // CS1579
      {
         Console.WriteLine(i);
      }
   }
}
```

Compiler Error CS1612

无法修改“expression”的返回值，因为它不是变量

尝试修改作为中间表达式的结果生成但未存储在变量中的值类型。如果尝试直接修改泛型集合中的结构，则会发生此错误，如下面的示例所示：

```
List<myStruct> list = {…};
list[0].Name = "MyStruct42"; //CS1612
```

若要修改结构，请首先将其赋给局部变量，修改该变量，然后将该变量赋回给集合中的项。

```
List<myStruct> list = {…};
MyStruct ms = list[0];
ms.Name = "MyStruct42";
list[0] = ms;
```

之所以会发生此错误，原因是在赋值时复制了值类型。在从属性或索引器中检索值类型时，所获取的是对象的副本，而不是对于对象本身的引用。由于属性或索引器实际上是方法，而不是存储位置（变量），因此它们不存储返回的副本。您必须将副本存储在声明的变量中，然后才能对其进行修改。

引用类型不会发生此错误，因为在这种情况下，属性或索引器将返回对现有对象的引用，而对象是存储位置。

如果是在定义类或结构，则修改属性声明，提供对结构成员的访问，即可解决此错误。如果是在编写客户端代码，则创建您自己的结构实例，修改其字段，然后将整个结构赋回给该属性，即可解决此错误。第三种替代方式是，您可以将结构更改为类。

当您尝试通过将返回整个结构的封闭类的属性来访问结构的成员时，也会发生 CS1612，如下面的示例所示：

```
// CS1612.cs
using System;

public struct MyStruct
{
    public int Width;
}

public class ListView
{
    MyStruct ms;
    public MyStruct Size
    {
        get { return ms; }
        set { ms = value; }
    }
}

public class MyClass
{
    public MyClass()
    {
        ListView lvi;
        lvi = new ListView();
        lvi.Size.Width = 5; // CS1612

        // You can use the following lines instead.
        // MyStruct ms;
        // ms.Width = 5;
        // lvi.Size = ms;  // CS1612
    }

    public static void Main()
    {
        MyClass mc = new MyClass();
        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

请参阅

结构（C# 编程指南）

值类型（C# 参考）

引用类型（C# 参考）

Compiler Error CS1614

“name”不明确的；在“attribute1”和“attribute2”之间。请使用"
@attribute”或“attributeAttribute”

编译器遇到了意义不明确的特性指定。

为简便起见，C# 编译器允许将 **ExampleAttribute** 指定为 [Example]。不过，如果名为 **Example** 的特性类与 **ExampleAttribute** 同时存在，则会引起多义性，因为编译器无法判断 [Example] 是指 **Example** 特性还是指 **ExampleAttribute** 特性。要解决此问题，请对 **Example** 特性使用 [@Example]，对 **ExampleAttribute** 使用 [ExampleAttribute]。

下面的示例生成 CS1614：

```
// CS1614.cs
using System;

// Both of the following classes are valid attributes with valid
// names (MySpecial and MySpecialAttribute). However, because the l
// rules for attributes involves auto-appending the 'Attribute' suf
// to the identifier, these two attributes become ambiguous; that i
// if you specify MySpecial, the compiler can't tell if you want
// MySpecial or MySpecialAttribute.

public class MySpecial : Attribute {
   public MySpecial() {}
}

public class MySpecialAttribute : Attribute {
   public MySpecialAttribute() {}
}

class MakeAWarning {
   [MySpecial()] // CS1614
                 // Ambiguous: MySpecial or MySpecialAttribute?
   public static void Main() {
   }

   [@MySpecial()] // This isn't ambiguous, it binds to the first at
   public static void NoWarning() {
   }

   [MySpecialAttribute()] // This isn't ambiguous, it binds to the
   public static void NoWarning2() {
   }

   [@MySpecialAttribute()] // This is also legal.
   public static void NoWarning3() {
   }
}
```

Compiler Error CS1640

foreach 语句不能对“type”类型的变量进行操作，因为它实现“interface”的多个实例化，请尝试强制转换为特定的接口实例化

此类型从<T> 的两个或更多的实例继承，这意味着不存在 **foreach** 可以使用的类型唯一枚举。指定 IEnumeratorr<T> 的类型，或使用其他循环构造。

下面的示例生成 CS1640：

```
// CS1640.cs

using System;
using System.Collections;
using System.Collections.Generic;

public class C : IEnumerable, IEnumerable<int>, IEnumerable<string>
{
    IEnumerator<int> IEnumerable<int>.GetEnumerator()
    {
        yield break;
    }

    IEnumerator<string> IEnumerable<string>.GetEnumerator()
    {
        yield break;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return (IEnumerator)((IEnumerable<string>)this).GetEnumerat
    }
}

public class Test
{
    public static int Main()
    {
        foreach (int i in new C()){}    // CS1640

        // Try specifing the type of IEnumerable<T>
        // foreach (int i in (IEnumerable<int>)new C()){}
        return 1;
    }
}
```

Compiler Error CS1644

功能“feature”不是标准化 ISO C# 语言规范的一部分，有些编译器可能不接受它

如果您指定了 /langversion 选项 ISO-1，而您编译的代码所使用的功能不是 ISO 1.0 标准的组成部分，则会发生此错误。若要解决此错误，请不要对任何 C# 2.0 编译器功能（如泛型或匿名方法）使用 ISO-1 兼容性选项。

下面的示例生成 CS1644：

```
// CS1644.cs
// compile with: /langversion:ISO-1 /target:library
class C<T> {}   // CS1644
```

Compiler Error CS1656

无法给“variable”赋值，因为它是“read-only variable type”

当变量赋值发生在只读上下文中时，将出现此错误。只读上下文包括 foreach 迭代变量、using 变量和 fixed 变量。若要解决此错误，请避免在 **using** 块、**foreach** 语句和 **fixed** 语句中给变量赋值。

以下示例生成错误 CS1656，因为它尝试替换 **foreach** 循环内的集合中的所有元素。解决此错误的一种方法是将 **foreach** 循环更改为 for 循环。另一种方法（此处未显示）是修改现有元素的成员；这可能对类起作用，但对结构不起作用。

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;

namespace CS1656_2
{

    class Book
    {
        public string Title;
        public string Author;
        public double Price;
        public Book(string t, string a, double p)
        {
            Title=t;
            Author=a;
            Price=p;

        }
    }

    class Program
    {
        private List<Book> list;
        static void Main(string[] args)
        {
            Program prog = new Program();
            prog.list = new List<Book>();
            prog.list.Add(new Book ("The C# Programming Language",
                                    "Hejlsberg, Wiltamuth, Golde",
                                     29.95));
            prog.list.Add(new Book ("The C++ Programming Language",
                                    "Stroustrup",
                                     29.95));
            prog.list.Add(new Book ("The C Programming Language",
                                    "Kernighan, Ritchie",
```

```
                                    29.95));
            foreach(Book b in prog.list)
            {
                // Cannot modify an entire element in a foreach loo
                // even with reference types.
                // Use a for or while loop instead
                if (b.Title == "The C Programming Language")
                    // Cannot assign to 'b' because it is a 'foreac
                    // iteration variable'
                    b = new Book("Programming Windows, 5th Ed.", "P
            }

            //With a for loop you can modify elements
            //for(int x = 0; x < prog.list.Count; x++)
            //{
            //    if(prog.list[x].Title== "The C Programming Langua
            //        prog.list[x] = new Book("Programming Windows,
            //}
            //foreach(Book b in prog.list)
            //    Console.WriteLine(b.Title);

        }
    }
}
```

下面的示例演示如何在除 **foreach** 循环之外的其他上下文中生成 CS1656：

```
// CS1656.cs
// compile with: /unsafe
using System;

class C : IDisposable
{
    public void Dispose() { }
}

class CMain
{
    unsafe public static void Main()
    {
        using (C c = new C())
        {
            // Cannot assign to 'c' because it is a 'using variable
            c = new C(); // CS1656
        }

        int[] ary = new int[] { 1, 2, 3, 4 };
        fixed (int* p = ary)
        {
            // Cannot assign to 'p' because it is a 'fixed variable
            p = null; // CS1656
        }
    }
}
```

Compiler Error CS1674

“T”：using 语句中使用的类型必须可以隐式转换为“System.IDisposable”

using 语句 旨在用于确保在 **using** 块的最后处置对象，因此只有可处置的类型才能在这类语句中使用。例如，值类型不是可处置的，并且没有约束为类的类型参数不能假定为可处置。

下面的示例生成 CS1674。

```
// CS1674.cs
class C
{
   public static void Main()
   {
      int a = 0;
      a++;

      using (a) {}   // CS1674
   }
}
```

下面的示例生成 CS1674。

```
// CS1674_b.cs
using System;
class C {
   public void Test() {
      using (C c = new C()) {}   // CS1674
   }
}

// OK
class D : IDisposable {
   void IDisposable.Dispose() {}
   public void Dispose() {}

   public static void Main() {
      using (D d = new D()) {}
   }
}
```

下面的示例说明了需要使用一个类类型约束来确保未知的类型参数是可释放的。下面的示例生成 CS1674。

```
// CS1674_c.cs
// compile with: /target:library
using System;
public class C<T>
// Add a class type constraint that specifies a disposable class.
// Uncomment the following line to resolve.
// public class C<T> where T : IDisposable
{
   public void F(T t)
   {
      using (t) {}   // CS1674
   }
}
```

Compiler Error CS1703

已导入了具有相同的简单名称“name”的程序集。请尝试移除其中一个引用，或者给引用加上签名以启用并列模式。

编译器移除有相同的路径和文件名称的引用，但有可能相同的文件在两个位置都存在，或者您忘了更改版本号。此错误指出两个引用具有相同的程序集标识，因此编译器无法在元数据中区分它们。或者移除其中的一个冗余引用，或者用某种办法使引用一致，例如通过提高程序集版本号。

以下代码生成错误 CS1703。

此代码在 .\bin1目录创建A程序集.

将此示例保存在名为 CS1703a1.cs 的文件中，并用以下标志编译该文件：/t:library /out:.\bin1\cs1703.dll /keyfile:key.snk

```
using System;
public class A { }
```

此代码在.\bin2目录创建A程序集。

将此示例保存在名为 CS1703a2.cs 的文件中，并用以下标志编译该文件：/t:library /out:.\bin2\cs1703.dll /keyfile:key.snk

```
using System;
public class A { }
```

此代码引用前面两个模块中的程序集 A。

将此示例保存在名为 CS1703ref.cs 的文件中，并用以下标志编译该文件：/t:library /r:A2=.\bin2\cs1703.dll /r:A1=.\bin1\cs1703.dll

```
extern alias A1;
extern alias A2;
```

Compiler Error CS1704

已导入具有相同简单名称“Assembly Name”的程序集。请尝试移除其中一个引用，或者给引用加上签名以启用并列模式。

此错误指出有两个引用具有相同的程序集标识，这是因为相关程序集缺少强名称而且没有签名，因而编译器无法在元数据中区分它们。因此，运行时会忽略版本和区域性程序集名称属性。用户应移除多余的引用，重命名其中一个引用，或者为它们提供一个强名称。

此示例创建一个程序集并将它保存到根目录下。

```
// CS1704_a.cs
// compile with: /target:library /out:c:\\cs1704.dll
public class A {}
```

此示例创建一个与上面的示例同名的程序集，但将它保存到不同的位置。

```
// CS1704_b.cs
// compile with: /target:library /out:cs1704.dll
public class A {}
```

此示例尝试引用这两个程序集。下面的示例生成 CS1704。

```
// CS1704_c.cs
// compile with: /target:library /r:A2=cs1704.dll /r:A1=c:\\cs1704
// CS1704 expected
extern alias A1;
extern alias A2;
```

Compiler Error CS1705

程序集“AssemblyName1”所使用的“TypeName”的版本高于所引用的程序集“AssemblyName2”的版本

您要进入一个版本号高于所引用的程序集的版本号的类型。此错误通常由意外使用相同程序集的两个版本引发。

例如，假设您有两个程序集，Asmb1 和 Asmb2。程序集 Asmb1 引用生成程序集 Asmb2 1.0。程序集 Asmb1 还使用已添加到 2.0 版的程序集 Asmb2 的类 MyClass。编译器有一致性绑定引用规则，对 2.0 版的MyClass的引用无法由 1.0 版满足。

下面详细示例由四个代码模块：

- 两个 DLL，它们除了版本特性外，其他方面都是一样的。
- 引用前两个的第三个 DLL。
- 引用相同的 DLL的1.0版本的客户，但是，从版本 2.0 访问的类。

下面的代码创建两个相同的 DLL 中的第一个。有关如何生成密钥文件的信息，请参见 /keyfile (C# Compiler Options)。

```
// CS1705a.cs

// Compile by using the following command:
//      csc /target:library /out:C:\\CS1705.dll /keyfile:mykey.snk

// The DLL is created in the C:\ directory.

// The AssemblyVersion attribute specifies version 1.0 for this DLL

[assembly:System.Reflection.AssemblyVersion("1.0")]
public class Class1
{
   public void Method1() {}
}
```

下面的代码定义是程序集 2.0 版，由特性 AssemblyVersionAttribute 指定。

```
// CS1705b.cs

// Compile by using the following command:
//     csc /target:library /out:CS1705.dll /keyfile:mykey.snk CS170

// The DLL is created in the current directory.

// The AssemblyVersion attribute specifies version 2.0 for this DLL

[assembly:System.Reflection.AssemblyVersion("2.0")]
public class Class1
{
    public void Method1() { }
}
```

下面的代码引用前面的代码中定义的两个 DLL 版本。 AssemblyA 引用 CS1705 a.cs (版本 1.0)创建的 DLL。 AssemblyB 引用 CS1705b.cs (版本 2.0)创建的 DLL。两个方法是在 ClassC 中定义的。第一，Return1A，它返回 Class1A类型的对象，是从 DLL 版本 1.0的Class1 的别名。第二，Return1B，它返回 Class1B类型的对象，是从 DLL 版本 2.0的Class1 的别名。 Return1A 的定义在版本 1.0 上创建一个依赖关系，这些依赖关系；Return1B 的定义在创建版本 2.0上创建一个依赖关系。

```
// CS1705c.cs

// Compile by using the following command. AssemblyA refers to the
// CS1705a.cs (version 1.0). AssemblyB refers to the DLL created by
// (version 2.0).
//    csc /target:library /r:AssemblyA=C:\\CS1705.dll /r:AssemblyB=

extern alias AssemblyA;
extern alias AssemblyB;

// Class1A is an alias for type Class1 from VS1705a.cs, which is in
// of the assembly. Class1B is an alias for type Class1 from CS1705
// is in version 2.0 of the assembly.

using Class1A = AssemblyA::Class1;
using Class1B = AssemblyB::Class1;

// Method Return1A in ClassC returns an object of type Class1A, whi
// Class1 from version 1.0 of the DLL. Method Return1B returns an o
// of type Class1B, which is Class1 from version 2.0 of the DLL.

public class ClassC
{
    // The following line creates a dependency on version 1.0 of CS
    // This is not the source of the problem when ClassC is accesse
    // CS1705d.cs because CS1705d.cs references version 1.0 of the
    // Therefore, type Class1A and the assembly have the same versi
    public static Class1A Return1A() { return new Class1A(); }

    // The following line creates a dependency on version 2.0 of CS
    // This causes compiler error CS1705 when ClassC is accessed fr
    // CS1705d.cs, because CS1705d.cs does not reference version 2.
    // DLL. Class1B is the alias for Class1 in version 2.0, and CS1
    // references version 1.0.
    public static Class1B Return1B() { return new Class1B(); }
}
```

下面的代码可生成编译器错误CS1705。它引用CS1705 a.cs (版本 1.0)创建的DLL。但是，在 Main 方法中，代码从CS1705 c.cs 的 ClassC来访问 。 ClassC 使用在 CS1705 b.cs(2.0 版)中定义的类型 。这导致编译器错误 CS1705，因为类型具有高于引用的CS1705.dll的版本号的CS1705 的 .dll 的版本号。

```
// CS1705d.cs

// Compile by using the following command:
//     csc /reference:C:\\CS1705.dll /reference:CS1705c.dll CS1705c

// C:\\CS1705.dll is version 1.0 of the assembly.

class Tester
{
    static void Main()
    {
        // Return1A returns a type defined in version 1.0.
        ClassC.Return1A().Method1();
        // Return1B returns a type defined in version 2.0.
        ClassC.Return1B().Method1();
    }
}
```

可以通过以下方式之一来解决 这个错误：

- 更新代码，以便所有 DLL 文件使用的版本相同。
- 使用以下命令编译，把对 DLL 2.0 版的引用添加到 d.cs CS1705:

  csc /reference:C:\CS1705.dll /reference:CS1705.dll /reference:CS1705c.dll CS1705d.cs

  虽然使用此命令时程序在编译，则在不运行。为了使程序在运行，则可提供包含使用<assemblyIdentity>和<codeBase> 子元素来指定 DLL1.0 版本位置的 <dependentAssembly> 元素的应用程序配置文件。有关配置文件的更多信息，请参见使用配置文件配置应用。

请参阅

外部别名（C# 参考）

:: 运算符（C# 参考）

Command-line Building With csc.exe

Compiler Error CS1708

固定大小的缓冲区只能通过局部变量或字段访问

C# 2.0 的一个新功能是能够在 **struct** 的内部定义内嵌数组。这类数组只能通过局部变量或字段访问，不能作为表达式左侧的中间值被引用。此外，这些数组无法通过 **static** 或 **readonly** 的字段访问。

要解决此错误，请定义一个数组变量，然后将内嵌数组分配给该变量。或者，从表示内嵌数组的字段中移除 **static** 或 **readonly** 修饰符。

下面的示例生成 CS1708。

```
// CS1708.cs
// compile with: /unsafe
using System;

unsafe public struct S
{
    public fixed char name[10];
}

public unsafe class C
{
    public S UnsafeMethod()
    {
        S myS = new S();
        return myS;
    }

    static void Main()
    {
        C myC = new C();
        myC.UnsafeMethod().name[3] = 'a';  // CS1708
        // Uncomment the following 2 lines to resolve:
        // S myS = myC.UnsafeMethod();
        // myS.name[3] = 'a';

        // The field cannot be static.
        C._s1.name[3] = 'a';  // CS1708

        // The field cannot be readonly.
        myC._s2.name[3] = 'a';  // CS1708
    }

    static readonly S _s1;
    public readonly S _s2;
}
```

Compiler Error CS1716

不要使用“System.Runtime.CompilerServices.FixedBuffer”特性。请改用“fixed”字段修饰符。

在包含与字段声明相似的固定大小的数组声明的不安全代码段中，会发生此错误。不要使用此特性。请改用关键字 **fixed**。

下面的示例生成 CS1716。

```
// CS1716.cs
// compile with: /unsafe
using System;
using System.Runtime.CompilerServices;

public struct UnsafeStruct
{
    [FixedBuffer(typeof(int), 4)]  // CS1716
    unsafe public int aField;
    // Use this single line instead of the above two lines.
    // unsafe public fixed int aField[4];
}

public class TestUnsafe
{
    static int Main()
    {
        UnsafeStruct us = new UnsafeStruct();
        unsafe
        {
            if (us.aField[0] == 0)
                return us.aField[1];
            else
                return us.aField[2];
        }
    }
}
```

编译器错误 CS1721

类“class”不能有多个基类：“class_1”和“class_2”

导致此错误的最常见的原因是尝试使用多重继承。 C# 中的类只能从一个类直接继承。 但是，类可以实现任意数量的接口。

示例

下面的示例演示生成 SC1721 的一种方式，然后演示避免该错误的两种可行方法。

```
// CS1721.cs
public class A {}
public class B {}
public class MyClass : A, B {}   // CS1721

// One possible fix is to use the following approach instead:
public class A {}
public class B : A {}
public class C : B {}

// Another possible fix is to use interfaces instead of base classe
public class A {}
public interface B {}
public class C : A, B {}
```

请参见

参考

多态性（C# 编程指南） 接口（C# 编程指南）

Compiler Error CS1726

友元程序集引用“reference”无效。强名称签名的程序集必须在其 InternalsVisibleTo 声明中指定一个公钥。

强名称签名的程序集只能将使用 InternalsVisibleToAttribute 创建的友元程序集访问权限授予其他强签名的程序集。

若要解决 CS1726，请对要授予其友元访问权限的程序集进行签名（赋予其强名称），或不授予友元访问权限。

有关更多信息，请参见友元程序集（C# 和 Visual Basic）。

下面的示例生成 CS1726。

```
// Save this code as CS1726.cs

// Run the following command to create CS1726.key:
//      sn -k CS1726.key

// Then compile by using the following command:
//      csc /keyfile:CS1726.key /target:library CS1726.cs

using System.Runtime.CompilerServices;

// The following line causes compiler error CS1726.
[assembly: InternalsVisibleTo("UnsignedAssembly")]

// To get rid of the error, try the following line instead.
//[assembly: InternalsVisibleTo("SignedAssembly, PublicKey=0024000(

class A { }
```

请参阅

如何：创建签名的友元程序集（C# 和 Visual Basic）

Compiler Error CS1729

“type”不包含采用“number”参数的构造函数。

在直接或间接调用类的构造函数时，如果编译器找不到具有相同数量参数的任意构造函数，则会发生此错误。在下面的示例中，test 类没有采用任意参数的构造函数。因此，该类只有不含参数的默认构造函数。由于在生成该错误的第二行上，派生类没有声明其自己的构造函数，因此编译器将提供默认构造函数。该构造函数在基类中调用无参数构造函数。由于基类没有此类构造函数，因此会生成 CS1729。

更正此错误

1. 调整构造函数调用中的参数数量。
2. 修改相应的类，提供包含必须调用的参数的构造函数。
3. 在基类中提供无参数构造函数。

下面的示例生成 CS1729：

```
// cs1729.cs
class Test
{
    static int Main()
    {
        // Class Test has only a default constructor, which takes n
        Test test1 = new Test(2); // CS1729
        // The following line resolves the error.
        Test test2 = new Test();

        // Class Parent has only one constructor, which takes two i
        Parent exampleParent1 = new Parent(10); // CS1729
        // The following line resolves the error.
        Parent exampleParent2 = new Parent(10, 1);

        return 1;
    }
}

public class Parent
{
    // The only constructor for this class has two parameters.
    public Parent(int i, int j) { }
}

// The following declaration causes a compiler error because class
// does not have a constructor that takes no arguments. The declara
// class Child2 shows how to resolve this error.
public class Child : Parent { } // CS1729

public class Child2 : Parent
{
    // The constructor for Child2 has only one parameter. To access
    // constructor in Parent, and prevent this compiler error, you
    // a value for the second parameter of Parent. The following ex
    public Child2(int k)
        : base(k, 0)
    {
        // Add the body of the constructor here.
    }
}
```

请参阅

继承（C# 编程指南）

构造函数（C# 编程指南）

Compiler Error CS1919

创建对象时不能使用不安全的类型“type name”。

new 运算符仅在托管堆上创建对象。但可以通过使用语言的互操作功能调用返回指针的本机方法，间接地在非托管内存中创建对象。

更正此错误

1. 在新的对象创建表达式中使用安全类型。例如，使用 **char** 或 **int**，而不是使用 **char*** 或 **int***。
2. 如果必须在非托管内存中创建对象，请使用 Win32 或 COM 方法，或者用 C 或 C++ 编写自己的函数并从 C# 调用该函数。

下面的示例生成 CS1919，因为指针类型是不安全的：

```
// cs1919.cs
// Compile with: /unsafe
unsafe public class C
{
    public static int Main()
    {
        var col1 = new int* { }; // CS1919
        var col2 = new char* { }; // CS1919
        return 1;
    }
}
```

请参阅

互操作性（C# 编程指南）

Compiler Error CS1921

与“method”最匹配的重载方法具有对于初始值设定项元素而言错误的签名。可初始化的 Add 必须是可访问的实例方法。

在尝试对没有公共非静态 **Add** 方法的类使用集合初始值设定项时，会生成此错误。如果 **Add** 方法因其保护级别（**private**、**protected** 和 **internal**）而不可访问，则会收到到 CS0122，因此该错误可能意味着该方法被定义为 **static**。

下面的示例生成 CS1921：

```
// cs1921.cs
using System.Collections;
public class C : CollectionBase
{
    public static void Add(int i)
    {
    }
}
public class Test
{
    public static void Main()
    {
        var collection = new C { 1, 2, 3 }; // CS1921
    }
}
```

请参阅

对象和集合初始值设定项（C# 编程指南）

Compiler Error CS1926

读取 Win32 清单文件“filename”时出错 --“error”。

如果满足以下条件，则会生成此错误：

1. 在命名行上或通过以下方法指定了 **/win32manifest** 选项：右击“解决方案资源管理器”中的“项目”图标，指向“添加”，单击“新建项”，然后单击“应用程序清单文件”。
2. 该文件损坏或缺失。

更正此错误

1. 移除该选项。
2. 替换、修复或重新生成该文件。

在使用损坏的 win32 清单文件或缺少该文件的情况下编译时，下面的示例会生成 CS1926：

```
// cs1926.cs
// Compile with: /win32manifest: ../../app.manifest
// CS1926
class Test
{
    public static int Main()
    {
        return 1;
    }
}
```

请参阅

/win32manifest (C# Compiler Options)

Compiler Error CS1933

表达式不能包含查询表达式

有些变量不能用查询表达式进行初始化。常量不能用查询表达式进行初始化，因为常量只能用文本、命名常量和数学运算符的某一组合进行初始化。

更正此错误

1. 从查询变量中移除该修饰符。

下面的示例生成 CS1933：

```
// cs1933.cs
using System.Linq;
using System.Collections;

class P
{
    const IEnumerable e = from x in new[] { 1, 2, 3 } select x; //
    static int Main()
    {
        return 1;
    }
}
```

请参阅

LINQ 查询表达式（C# 编程指南）

Compiler Error CS1936

未能找到源类型“<类型>”的查询模式的实现。“方法”未找到。

为了查询某一源类型，该类型必须实现要在查询中调用的标准查询运算符方法。该实现可以采用通过适当的 **using** 指令放入范围中的类成员或扩展方法的形式。

更正此错误

- 请确保查询的是对象集合，而不是单个对象。
- 请确保指定了所需 **using** 指令。

下面的示例生成 CS1936：

```
// cs1936.cs
using System.Collections;
using System.Linq;
class Test
{
    static int Main()
    {
        object obj;
        IEnumerable e = from x in obj // CS1936
                        select x;
        return 0;
    }
}
```

如果无意中尝试查询某一类型的单个对象而不是此类对象的集合，则通常会发生此错误。

请参阅

Standard Query Operators Overview

Compiler Error CS1941

“clause”子句中一个表达式的类型不正确。类型推理在对“method”的调用中失败。

查询表达式中的类型推理源于数据源中的元素类型。

更正此错误

1. 如果发生此错误的原因不十分明显，请仔细检查查询，并跟踪从数据源到错误发生点的每个子句的结果的类型。

下面的代码生成 CS1941，因为 **equals** 运算符被要求对 **int** 和 **string** 进行比较。

```
// cs1941.cs
using System.Collections;
using System.Linq;
class Test
{
    static int Main()
    {
        var nums = new[] { 1, 2, 3, 4, 5, 6 };
        var words = new string[] { "lake", "mountain", "sky" };
        IEnumerable e = from n in nums
                        join w in words on n equals w // CS1941
                        select w;
        return 0;
    }
}
```

发生类型推理失败的方法是查询子句在编译时转换为的方法。

请参阅

LINQ 查询表达式（C# 编程指南）

Type Relationships in LINQ Query Operations (C#)

Compiler Error CS1942

“clause”子句中表达式的类型不正确。类型推理在对“method”的调用中失败。

如果为范围变量指定了不正确的显式类型，通常会生成此错误。

更正此错误

1. 如果范围变量是显式类型的，请确保该类型与循环访问的集合中的元素类型相同或可从该类型隐式转换。如果范围变量的前面带有 **var** 关键字，请移除 **var**。

下面的代码生成 CS1942：

```
// cs1942.cs
class Program
    {
        static void Main(string[] args)
        {
            var x = from var i in Enumerable.Range(1, 100) // CS194
                    select i; //CS1942
        }
    }
```

CS1942 与 CS1949 相关，因为对范围变量使用 **var** 会导致因 **var** 不是类型而使基础 Cast<T> 运算失败。

请参阅

var（C# 参考）

LINQ 查询表达式（C# 编程指南）

Compiler Error CS1943

在具有源类型“type”的查询表达式中后面的 from 子句中不允许使用类型为“type”的表达式。类型推理在对“method”的调用中失败。

所有范围变量都必须表示可查询的类型。

更正此错误

1. 请确保该类型是实现 **IEnumerable**、**IEnumerable<T>** 或派生接口的可查询类型，或者是定义了查询模式的任何其他类型。

2. 如果该类型是非泛型 **IEnumerable**，请在范围变量上提供显式类型。

下面的代码生成 CS1943：

```
// cs1943.cs
using System.Linq;
class Test
{
    class TestClass
    { }
    static void Main()
    {
        int[] nums = { 0, 1, 2, 3, 4, 5 };
        TestClass tc = new TestClass();

        var x = from n in nums
                from s in tc // CS1943
                select n + s;
    }
}
```

Compiler Error CS1946

无法将匿名方法表达式转换为表达式树。

匿名方法表示一组语句，而表达式树不能包含任何语句。因此，匿名方法不能由表达式树表示。

更正此错误

1. 将匿名方法更改为 lambda 表达式。

下面的示例生成 CS1946：

```
// cs1946.cs
using System;
    using System.Linq.Expressions;

    public delegate void D();

    class Test
    {
        static void Main()
        {
            Expression<D> tree = delegate() { }; //CS1946
            // Try using a lambda expression instead.
            // Expression<D> tree = (x) => x + 1;
        }
    }
```

请参阅

匿名方法（C# 编程指南）

表达式树（C# 和 Visual Basic）

编译器错误 CS2032

不允许命令行上或响应文件中存在字符“character”

csc.exe 的命令行选项和响应文件不允许包含 0-31 范围内的 ASCII 码控制字符或者管道 (|) 字符。

编译器错误 CS2032无法从命令行演示，因为命令行处理器和集成开发环境 (IDE) 将筛选出无效的字符。 使用 响应文件，下面的过程将生成错误。

生成此错误

1. 在 我的文档 文件夹中，创建一个名为 CS2032 .rsp 的文本文件，然后使中的以下编译器选项：

   ```
   /target:exe /out:cs|2032.exe cs2032.cs
   ```

2. 在 我的文档 文件夹中，创建一个名为 cs2032.cs，其中包含的文本文件需要的内容。
3. 打开“开始”、然后选择“所有程序”、“Microsoft Visual Studio 2010”、“Visual Studio 工具”，然后单击“Microsoft Visual Studio 命令提示(2010)”。

   “Visual Studio 命令提示”窗口打开。

4. 在 Visual Studio命令行提示窗口中，将当前的目录更改为 C:\Users\YourUsername\文档。
5. 从“Visual Studio 命令提示”中运行以下命令：csc @cs2032.rsp
6. 因为响应文件，CS2032 .rsp，出现 CS2032 错误信息，其中包含一个竖线，。

Compiler Warning (level 1) CS0420

“identifier”：对 volatile 字段的引用不被视为 volatile

volatile 字段不能像正常情况那样使用 **ref** 或 **out** 参数进行传递，因为它在函数的范围内不会被视为 volatile。这也有例外情况，例如当调用互锁 API 时。对于任何警告，您可以使用 #pragma warning 在您特意使用 volatile 字段作为引用参数的情况（这种情况很少出现）中禁用此警告。

下面的示例生成 CS0420：

```
// CS0420.cs
// compile with: /W:1
using System;

class TestClass
{
   private volatile int i;

   public void TestVolatile(ref int ii)
   {
   }

   public static void Main()
   {
      TestClass x = new TestClass();
      x.TestVolatile(ref x.i);   // CS0420
   }
}
```

Compiler Warning (level 1) CS0465

引入“Finalize”方法会妨碍析构函数调用。是否希望声明析构函数？

当您使用签名为 public virtual void Finalize 的方法创建类时，将发生此警告。

如果将这种类用作基类并且派生类定义一个析构函数，则该析构函数将重写基类的 Finalize 方法，而不是 Finalize。

下面的示例生成 CS0465。

```
// CS0465.cs
// compile with: /target:library
class A
{
   public virtual void Finalize() {}   // CS0465
}

// OK
class B
{
   ~B() {}
}
```

Compiler Warning (level 1) CS1058

前一个 catch 子句已经捕获所有异常。引发的所有异常均被包装在 System.Runtime.CompilerServices.RuntimeWrappedException 中

如果 catch() 块在 catch (System.Exception e) 块后没有指定的异常类型，则该特性会导致 CS1058。该警告建议 catch() 块将不捕获任何异常。

如果在 AssemblyInfo.cs 文件中将 **RuntimeCompatibilityAttribute** 设置为 False，则 catch (System.Exception e) 块之后的 catch() 块可捕获非 CLS 异常：[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]。如果未将此特性显式设置为 False，则所有引发的非 CLS 异常将包装为异常，并且 catch (System.Exception e) 块将捕获它们。有关更多信息，请参见如何：捕捉非 CLS 异常。

下面的示例生成 CS1058。

```
// CS1058.cs
// CS1058 expected
using System.Runtime.CompilerServices;

// the following attribute is set to true by default in the C# comp
// set to false in your source code to resolve CS1058
[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = t

class TestClass
{
   static void Main()
   {
      try {}

      catch (System.Exception e) {
         System. Console.WriteLine("Caught exception {0}", e);
      }

      catch {}   // CS1058\. This line will never be reached.
   }
}
```

Compiler Warning (level 1) CS1060

使用了可能未赋值的字段“name”。如果结构未赋值，则结构实例变量在初始化时将不被赋值。

在未进行显式初始化的情况下，结构成员将初始化为其默认值。类类型（及其他引用类型）的默认值为 null。如果在没有对类进行初始化的情况下就尝试访问它，则会在运行时引发 **NullReferenceException**。编译器无法最终确定类成员是否将进行初始化，因此 CS1060 是警告而不是错误。

更正此错误

1. 为 **struct** 提供一个初始化其所有成员的构造函数。

下面的代码生成 CS1060，因为类类型 U 是 struct S 的成员但从未初始化。

```
// cs1060.cs
namespace CS1060
{
    public class U
    {
        public int i;
    }

    public struct S
    {
        public U u;
        // Add constructor to correct the error.
        //public S(int val)
        //{
        //    u = new U() { i = val };
        //}
    }
    public class Test
    {
        static void Main()
        {
            S s;
            s.u.i = 5;  // CS1060

            //Try these lines instead, and uncomment the constructo
            // S s = new S(0);
            // s.u.i = 5;

        }
    }
}
```

请参阅

结构（C# 编程指南）

Compiler Warning (level 1) CS1598

未能加载 XML 分析器，原因如下：“<原因>”。将不生成 XML 文档文件“<文件>”。

指定了 /doc 选项，但编译器未能找到并加载 msxml3.dll。请确保 msxml3.dll 文件已安装并注册。

Compiler Warning (level 1) CS1607

程序集生成 — reason

在编译的程序集创建阶段生成了警告。

如果要在 32 位操作系统上生成一个 64 位应用程序，必须确保在目标操作系统上安装了所有引用的程序集的 64 位版本。

所有特定于 x86 的公共语言运行时 (CLR) 程序集都具有 64 位的对应项（每个 CLR 程序集都存在于所有操作系统上）。因此，对于 CLR 程序集可以放心地忽略 CS1607。

如果在创建 AssemblyInformationalVersionAttribute 时遇到此警告，则可忽略此警告。信息性版本是一个字符串，它将附加的版本信息附加到一个程序集；此信息不在运行时使用。虽然可以指定任意文本，但是如果字符串的格式不是程序集版本号使用的格式，或者虽然是这种格式但包含通配符，则在编译时会显示一条警告消息。此警告无碍。

有关更多信息，请参见 Al.exe 工具错误和警告。

Compiler Warning (level 1) CS1616

选项“option”重写源文件或附加模块中给出的特性“attribute”

如果源文件中的程序集特性 AssemblyKeyFileAttribute 或 AssemblyKeyNameAttribute 与项目属性中指定的 /keyfile 或 /keycontainer 命令行选项或密钥文件名或密钥容器冲突，则会出现此警告。

在下面的示例中，假定您有一个名为 cs1616.snk 的密钥文件。该文件可以用命令行生成：

```
sn -k CS1616.snk
```

下面的示例生成 CS1616：

```
// CS1616.cs
// compile with: /keyfile:cs1616.snk
using System.Reflection;

// To fix the error, remove the next line
[assembly: AssemblyKeyFile("cs1616b.snk")]  // CS1616

class C
{
  public static void Main()
  {
  }
}
```

Compiler Warning (level 1) CS1658

“warning text”。另请参见错误“error code”

编译器在用警告重写错误时会发出此警告。有关此问题的信息，请参考上面所提到的错误。若要从 Visual Studio IDE 内查找相应的错误，请使用索引。例如，如果上面的文本是“另请参见错误‘CS1037’”，则请在索引中查找 CS1037。

下面的示例生成 CS1658。

```
// CS1658.cs
// compile with: /doc:x.xml
// CS1584 expected
/// <summary>
/// </summary>
public class C
{
    /// <see cref="C.F(params object[])"/>  // CS1658
    public static void M()
    {
    }

    /// <summary>
    /// </summary>
    public void F(params object[] o)
    {
    }

    static void Main()
    {
    }
}
```

Compiler Warning (level 1) CS1683

对类型“Type Name”的引用声称在此程序集中定义了该类型，但源代码或任何添加的模块中并未定义该类型

当您导入的程序集反过来包含对您当前正在编译的程序集的引用，而正在编译的程序集不包含任何与该引用匹配的内容时，会出现此错误。导致此情况的一种操作是，编译您的程序集，它最初确实包含所导入的程序集引用的成员。然后更新您的程序集，不小心移除了导入的程序集所引用的成员。

Compiler Warning (level 1) CS1685

预定义类型“System.type name”是在全局别名的多个程序集中定义的；请使用“文件名”中的定义

当在两个程序集中都找到一个预定义系统类型（例如 System.int32）时，会出现此错误。如果您从两个不同的位置引用 mscorlib（例如尝试并列运行 1.0 和 1.1 版的 .NET Framework），会发生此错误。

编译器仅使用其中一个程序集中的定义。编译器只搜索全局别名，不搜索定义 **/reference** 的库。如果您指定了 **/nostdlib**，编译器将搜索 Object，并且以后所有预定义类型的搜索都将在包含 Object 的文件中进行。

Compiler Warning (level 1) CS1690

由于“member”是引用封送类的字段，访问上面的成员可能导致运行时异常

当您尝试对从 MarshalByRefObject 派生的类成员调用方法、属性或索引器时，并且该成员是值类型时，会出现此警告。从 **MarshalByRefObject** 继承的对象通常应由跨应用程序域的引用进行封送。如果任何代码曾经尝试跨应用程序域直接访问的此类对象的值类型成员，则会发生运行时异常。若要解决此警告，请先将该成员复制到局部变量并对该变量调用方法。

下面的示例生成 CS1690：

```
// CS1690.cs
using System;

class WarningCS1690: MarshalByRefObject
{
   int i = 5;

   public static void Main()
   {
     WarningCS1690 e = new WarningCS1690();
     e.i.ToString();   // CS1690

     // OK
     int i = e.i;
     i.ToString();
     e.i = i;
   }
}
```

Compiler Warning (level 1) CS1691

“number”不是有效的警告编号

传递给 #pragma warning 预处理器指令的数字不是有效的警告编号。请验证该数字表示警告，而不是错误或其他字符序列。

下面的示例生成 CS1691。

```
// CS1691.cs
public class C
{
    int i = 1;
    public static void Main()
    {
        C myC = new C();
#pragma warning disable 151  // CS1691
// Try the following line instead:
// #pragma warning disable 1645
        myC.i++;
#pragma warning restore 151  // CS1691
// Try the following line instead:
// #pragma warning restore 1645
    }
}
```

Compiler Warning (level 1) CS1699

使用命令行选项“compiler_option”或适当项目设置代替“attribute_name”

为给程序集签名，必须指定密钥文件。在 Microsoft Visual C# 2005 之前，您在源代码中使用 CLR 特性指定了密钥文件。现在，这些特性都被否决了。

从 Microsoft Visual C# 2005 开始，您应使用“项目设计器”的“签名页”或程序集链接器指定密钥文件。

“项目设计器”的“签名页”是首选方法；有关更多信息，请参见 “项目设计器”->“签名”页 和 管理程序集签名和清单签名。

如何：使用强名称为程序集签名 使用如下编译器选项：

- /keyfile (C# Compiler Options) 代替 AssemblyKeyFileAttribute 特性。
- /keycontainer (C# Compiler Options) 而不是AssemblyKeyNameAttribute。
- /delaysign (C# Compiler Options) 代替 AssemblyDelaySignAttribute。

否决这些特性的原因是：

- 存在安全问题，因为这些特性嵌入在由编译器生成的二进制文件中。因此，每个拥有您的二进制文件的人都会获得存储在其中的密钥。
- 存在可用性问题，因为特性中指定的路径是相对于当前工作目录（此目录在集成开发环境 (IDE) 中可能会发生更改）或输出目录的。因此，大多数情况下密钥文件有可能是..\..\mykey.snk。特性还增加了项目系统为附属程序集正确签名的难度。当您使用编译器选项代替这些特性时，可以使用密钥的完全限定路径和文件名，而无需在输出文件中嵌入任何内容；项目系统和源代码管理系统可在项目移动时正确处理该完整路径；项目系统可保留密钥文件的项目相对路径，同时仍将完整路径传递给编译器；其他生成程序可通过直接将适当的路径传递给编译器（而不是用正确的特性生成源文件）更加容易地为输出签名。
- 将特性用于友元程序集会影响编译器的效率。当您使用特性时，编译器在必须决定是否授予友元关系时不知道密钥是什么，因此它必须进行推测。在编译的最后，一旦编译器最后知道了密钥，它就能够对推测进行验证。当使用编译器选项指定密钥文件时，编译器可立即决定是否授予友元关系。

下面的示例生成 CS1699。若要解决此错误，请移除该特性并使用 **/delaysign** 进行编译。

```
// CS1699.cs
// compile with: /target:library
[assembly:System.Reflection.AssemblyDelaySign(true)]   // CS1699
```

请参阅

“项目设计器”->“签名”页

管理程序集签名和清单签名

如何：使用强名称为程序集签名

Compiler Warning (level 1) CS1762

由于程序集'<assembly2>'间接引用了程序集'<assembly1>'，一个引用将被创建并嵌入到互操作程序集 assembly1 中.请考虑更改任一程序集的“嵌入互操作类型”属性。

您已添加了一个对 **Embed Interop Types** 属性设置为 **True** 的程序集 (assembly1) 的引用。这将指示编译器嵌入该程序集中的互操作类型信息。但是，编译器无法嵌入该程序集中的互操作类型信息，因为您已引用的另一个程序集 (assembly2) 也引用了该程序集 (assembly1) 并且它的 **Embed Interop Types** 属性设置为 **False**。

注意
将某个程序集引用的 **Embed Interop Types** 属性设置为 **True** 相当于通过使用命令行编译器的 **/link** 选项来引用该程序集。

处理此警告

- 若要嵌入这两个程序集的互操作类型信息，请将对 assembly1 的所有引用的 **Embed Interop Types** 属性设置为 **True**。有关如何设置此属性的更多信息，请参见 演练：嵌入托管程序集中的类型（C# 和 Visual Basic）。
- 若要移除此警告，您可以将 assembly1 的 **Embed Interop Types** 属性设置为 **False**。在这种情况下，互操作类型信息由主互操作程序集 (PIA) 提供。

请参阅

/link (C# Compiler Options)

Programming with Primary Interop Assemblies

Compiler Warning (level 1) CS1956

成员“name”实现类型“type”中的接口成员“name”。在运行时有多个与该接口成员相匹配的项。将调用哪个方法取决于具体的实现。

如果两个接口方法的唯一区别在于：某个特定参数被**ref** 还是 **out**标记，则将生成此警告。最好更改代码以避免此警告，因为在运行时实际将调用哪个方法并不十分明显，或者无法保证在运行时将调用哪个方法。

尽管 C# 可以区分 **out** 和**ref**，CLR 却无法区分它们。决定实现接口的方法时，CLR 只是从中任选一个。

避免此警告

1. 为编译器提供某种方式来区分这两个方法。例如，可以为这两个方法提供不同的名称，或者为其中一个方法提供附加参数。

下面的代码将生成 CS1956，因为 Base 中这两个 Test 方法的唯一区别在于它们的第一个参数上的 ref/out 修饰符：

```
// cs1956.cs
class Base<T, S>
{
    // This is the method that should be called.
    public virtual int Test(out T x) // CS1956
    {
        x = default(T);
        return 0;
    }

    // This is the "last" method and is the one that ends up being
    public virtual int Test(ref S x)
    {
        return 1;
    }
}

interface IFace
{
    int Test(out int x);
}

class Derived : Base<int, int>, IFace
{
    static int Main()
    {
        IFace x = new Derived();
        int y;
        return x.Test(out y);
    }
}
```

Compiler Warning (level 1) CS3003

“变量”的类型不符合 CLS

public、protected 或 **protected internal** 变量的类型必须符合公共语言规范 (CLS)。有关 CLS 遵从性的详细信息，请参阅语言独立性和与语言无关的组件。

下面的示例生成 CS3003：

```
// CS3003.cs

[assembly:System.CLSCompliant(true)]
public class a
{
    public ushort a1;   // CS3003, public variable
    public static void Main()
    {
    }
}
```

Compiler Warning (level 1) CS3007

只是未命名数组类型不同的重载方法“method”不符合 CLS。

如果你有一个接受交错数组的重载方法，并且方法签名的唯一区别是数组的元素类型，则将发生此错误。为避免此错误，可考虑使用矩形数组而不是交错数组，使用一个额外的参数消除函数调用的歧义，重命名一个或多个重载方法，或者如果不需要 CLS 遵从性，则移除 CLSCompliantAttribute 特性。有关 CLS 遵从性的详细信息，请参阅语言独立性和与语言无关的组件。

下面的示例生成 CS3007：

```
// CS3007.cs
[assembly: System.CLSCompliant(true)]
public struct S
{
    public void F(int[][] array) { }
    public void F(byte[][] array) { }  // CS3007
    // Try this instead:
    // public void F1(int[][] array) {}
    // public void F2(byte[][] array) {}
    // or
    // public void F(int[,] array) {}
    // public void F(byte[,] array) {}
}
```

Compiler Warning (level 1) CS3009

“type”：基类型“type”不符合 CLS

在标记为符合公共语言规范 (CLS) 的程序集中，基类型被标记为不必符合 CLS。请移除指定程序集符合 CLS 的特性，或移除指示类型不符合 CLS 的特性。有关 CLS 遵从性的更多信息，请参见编写符合 CLS 的代码和语言独立性和与语言无关的组件。

下面的示例生成 CS3009：

```
// CS3009.cs

using System;

[assembly:CLSCompliant(true)]
[CLSCompliant(false)]
public class B
{
}

public class C : B   // CS3009
{
    public static void Main () {}
}
```

编译器警告（等级 1）CS4014

由于此调用不会等待，因此在调用完成前将继续执行当前方法。请考虑将“await”运算符应用于调用结果。

当前方法调用返回 Task 或 Task<TResult>，而不应用运算符 等候 到结果的异步方法。异步方法的调用启动异步任务。但是，因为 **await** 运算符没有应用，程序继续运行，而不必等待任务完成。在大多数情况下，此行为不是您所期望的。通常调用方法的其他特性取决于调用的结果，或最低限度上，调用方法的预计在从包含调用的方法返回之前完成。

同样至关重要的问题是被调用的异步方法中引发的异常发生了什么。在返回Task或Task<TResult>方法中引发的异常被存储在返回的任务中。如果不等待任务还不显式检查异常，该异常会丢失。如果在等待任务，异常会被重新抛出。

作为最佳做法，应始终在等待调用。

应考虑禁止显示警告，仅当您确定不希望等待异步调用完成并且这个被调用的方法不会引发任何异常。在此情况下，可以通过分配调用的任务结果给变量来禁止警告。

下面的示例演示如何生成警告，如何显示它以及如何等待调用。

```
async Task CallingMethodAsync()
{
    resultsTextBox.Text += "\r\n  Entering calling method.";
    // Variable delay is used to slow down the called method so tha
    // distinguish between awaiting and not awaiting in the progran
    // You can adjust the value to produce the output that this top
    // after the code.
    var delay = 5000;

    // Call #1.
    // Call an async method. Because you don't await it, its comple
    // isn't coordinated with the current method, CallingMethodAsyn
    // The following line causes warning CS4014.
    CalledMethodAsync(delay);

    // Call #2.
    // To suppress the warning without awaiting, you can assign the
    // returned task to a variable. The assignment doesn't change h
    // the program runs. However, recommended practice is always to
    // await a call to an async method.

    // Replace Call #1 with the following line.
    //Task delayTask = CalledMethodAsync(delay);

    // Call #3
    // To contrast with an awaited call, replace the unawaited call
    // (Call #1 or Call #2) with the following awaited call. Best
    // practice is to await the call.

    //await CalledMethodAsync(delay);

    // If the call to CalledMethodAsync isn't awaited, CallingMetho
    // continues to run and, in this example, finishes its work and
    // to its caller.
    resultsTextBox.Text += "\r\n  Returning from calling method.";
}

async Task CalledMethodAsync(int howLong)
{
    resultsTextBox.Text +=
        "\r\n    Entering called method, starting and awaiting Task

    // Slow the process down a little so that you can distinguish b
    // awaiting and not awaiting in the program's output. Adjust th
    // for howLong if necessary.
    await Task.Delay(howLong);
    resultsTextBox.Text +=
        "\r\n    Task.Delay is finished--returning from called meth
}
```

在此示例中，如果您调用Call #1或Call #2，在调用方 (CallingMethodAsync) 和调用方的调用方 (startButton_Click) 完成后，不等待异步方法 (CalledMethodAsync)。当调用的方法完成时，以下输出的最后一行向你演示了。从在完整的示例中调用 CallingMethodAsync 的事件处理程序，为输入和输出做标记。

```
Entering the Click event handler.
  Entering calling method.
    Entering called method, starting and awaiting Task.Delay.
  Returning from calling method.
Exiting the Click event handler.
    Task.Delay is finished--returning from called method.
```

使用 #pragma warning（C# 参考） 指令，还可以取消编译器警告。

以下 Windows Presentation Foundation (WPF) 应用程序包含从前面的方法。下列步骤建立应用程序。

1. 创建 WPF 应用程序，将其命名为 **AsyncWarning**。
2. 在 Visual Studio 代码编辑器中，选择“MainWindow.xaml”选项卡。

   如果此选项卡不可视，则在“解决方案资源管理器”中，打开 MainWindow.xaml 的快捷菜单，然后选择“查看代码”。

3. 在 MainWindow.xaml 的“XAML”视图中，使用下面的代码替换代码。

```
&lt;Window x:Class="AsyncWarning.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pre
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525"&gt;
    &lt;Grid&gt;
        &lt;Button x:Name="startButton" Content="Start" Horizor
        &lt;TextBox x:Name="resultsTextBox" Margin="0,80,0,0" T
    &lt;/Grid&gt;
&lt;/Window&gt;
```

   包含按钮和文本框的简单窗口显示在 MainWindow.xaml 的“设计”窗口中。

   有关 XAML设计器的更多信息，请参见在 Visual Studio 中，使用 XAML 设计器创建 UI。有关如何生成自己的简单的 UI，请参见 演练：使用 Async 和 Await 访问 Web（C# 和 Visual Basic）的“创建 WPF 应用程序”和“设计的简单 WPF MainWindow”一节。

4. 将MainWindow.xaml.cs中的代码替换为以下代码。

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace AsyncWarning
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, Rou
        {
            resultsTextBox.Text += "\r\nEntering the Click ever
            await CallingMethodAsync();
            resultsTextBox.Text += "\r\nExiting the Click event
        }

        async Task CallingMethodAsync()
        {
            resultsTextBox.Text += "\r\n  Entering calling meth
            // Variable delay is used to slow down the called m
            // distinguish between awaiting and not awaiting ir
            // You can adjust the value to produce the output t
            // after the code.
            var delay = 5000;

            // Call #1.
            // Call an async method. Because you don't await it
            // isn't coordinated with the current method, Calli
            // The following line causes warning CS4014.
            CalledMethodAsync(delay);

            // Call #2.
            // To suppress the warning without awaiting, you ca
            // returned task to a variable. The assignment does
            // the program runs. However, recommended practice 
            // await a call to an async method.

            // Replace Call #1 with the following line.
            //Task delayTask = CalledMethodAsync(delay);
```

```
            // Call #3
            // To contrast with an awaited call, replace the ur
            // (Call #1 or Call #2) with the following awaited
            // practice is to await the call.

            //await CalledMethodAsync(delay);

            // If the call to CalledMethodAsync isn't awaited,
            // continues to run and, in this example, finishes
            // to its caller.
            resultsTextBox.Text += "\r\n  Returning from callir
        }

        async Task CalledMethodAsync(int howLong)
        {
            resultsTextBox.Text +=
                "\r\n    Entering called method, starting and a

            // Slow the process down a little so that you can c
            // awaiting and not awaiting in the program's outpu
            // for howLong if necessary.
            await Task.Delay(howLong);
            resultsTextBox.Text +=
                "\r\n    Task.Delay is finished--returning from
        }
    }

    // Output with Call #1 or Call #2\. (Wait for the last line

    // Entering the Click event handler.
    //   Entering calling method.
    //     Entering called method, starting and awaiting Task.D
    //   Returning from calling method.
    // Exiting the Click event handler.
    //     Task.Delay is finished--returning from called methoc

    // Output with Call #3, which awaits the call to CalledMeth

    // Entering the Click event handler.
    //   Entering calling method.
    //     Entering called method, starting and awaiting Task.D
    //     Task.Delay is finished--returning from called methoc
    //   Returning from calling method.
    // Exiting the Click event handler.
}
```

5. 选择 F5 键运行程序，然后选择“开始”按钮。

   预期的输出显示在代码结尾。

请参阅

await（C# 参考）

使用 Async 和 Await 的异步编程（C# 和 Visual Basic）

Compiler Warning (level 2) CS0108

“member1”会隐藏继承的成员“member2”。如果打算隐藏，请使用 new 关键字。

声明变量所使用的名称与基类中的变量相同。但没有使用 new 关键字。此警告通知您应使用 **new**；声明变量时将按已在声明中使用了 **new** 来处理。

下面的示例生成 CS0108：

```
// CS0108.cs
// compile with: /W:2
using System;

namespace x
{
   public class clx
   {
      public int i = 1;
   }

   public class cly : clx
   {
      public static int i = 2;    // CS0108, use the new keyword
      // Use the following line instead:
      // public static new int i = 2;

      public static void Main()
      {
         Console.WriteLine(i);
      }
   }
}
```

请参阅

new 修饰符（C# 参考）

new

编译器警告（等级 2）CS0467

方法“method”和非方法“non-method”之间存在多义性。正在使用方法组。

如果继承的成员具有相同的签名，但来自不同的接口，则会导致多义性错误。

下面的示例会产生 CS0467。

```
// CS0467.cs
interface IList
{
    int Count { get; set; }
}

interface ICounter
{
    void Count(int i);
}

interface IListCounter : IList, ICounter {}

class Driver
{
    void Test(IListCounter x)
    {
        // The following line causes the warning. The assignment al
        // causes an error because you can't assign a value to a me
        x.Count = 1;
        x.Count(3);
        // To resolve the warning, you can change the name of the m
        // the property.

        // You also can disambiguate by specifying IList or ICounte
        ((IList)x).Count = 1;
        ((ICounter)x).Count(3);
    }

    static void Main()
    {
    }
}
```

Compiler Warning (level 2) CS0618

“member”已过时：“text”

类成员是用 **Obsolete** 特性标记的，以致在引用该类成员时会发出警告。有关更多信息，请参见常用特性（C# 和 Visual Basic）。

下面的示例生成 CS0618：

```
// CS0618.cs
// compile with: /W:2
using System;

public class C
{
   [Obsolete("Use newMethod instead", false)]   // warn if referenc
   public static void m2()
   {
   }

   public static void newMethod()
   {
   }
}

class MyClass
{
   public static void Main()
   {
      C.m2();  // CS0618
   }
}
```

Compiler Warning (level 2) CS1701

假定程序集引用“Assembly Name #1”与“Assembly Name #2”匹配，您可能需要提供运行时策略

这两个程序集在发行号和/或版本号上有差异。为保证一致，必须在应用程序的 .config 文件中指定指令，并提供程序集的正确强名称，如以下代码示例所示。

下面的多文件示例使用两个不同的外部别名引用一个程序集。第一个示例生成用于创建程序集 CS1701_d 的代码的较早版本。

```
// CS1701_a.cs
// compile with: /target:library /out:cs1701_d.dll /keyfile:mykey.s
using System.Reflection;
[assembly:AssemblyVersion("1.0")]
public class A {
   public void M1() {}
}

public class C1 {}
```

这是用于创建较新版本的程序集 CS1701_d 的代码。注意，由于这两个版本的输出文件同名，因此它需要将较新版本编译到不同于较早版本的目录中。

```
// CS1701_b.cs
// compile with: /target:library /out:c:\\cs1701_d.dll /keyfile:myk
using System.Reflection;
[assembly:AssemblyVersion("2.0")]
public class A {
   public void M2() {}
   public void M1() {}
}

public class C2 {}
public class C1 {}
```

此示例设置外部别名 A1 和 A2。

```
// CS1701_c.cs
// compile with: /target:library /reference:A2=c:\\cs1701_d.dll /re

extern alias A1;
extern alias A2;
// using System;
using a1 = A1::A;
using a2 = A2::A;

public class Ref {
   public static a1 A1() { return new a1(); }
   public static a2 A2() { return new a2(); }

   public static A1::C1 M1() { return new A1::C1(); }
   public static A2::C2 M2() { return new A2::C2(); }
}
```

此示例使用 A 的两个不同别名调用方法。下面的示例生成 C1701。

```
// CS1701_d.cs
// compile with: /reference:c:\\CS1701_d.dll /reference:CS1701_c.dl
// CS1701 expected
class Tester {
   public static void Main() {
      Ref.A1().M1();
      Ref.A2().M2();
   }
}
```

Compiler Warning (level 3) CS0675

按位“或”运算符在带符号扩展操作数上使用；请考虑首先强制转换为较小的无符号类型

编译器隐式地拓宽了并带符号扩展了变量，然后在按位“或”运算中使用了结果值。这可能导致意外的行为。

下面的示例生成 CS0675：

```
// CS0675.cs
// compile with: /W:3
using System;

public class sign
{
   public static void Main()
   {
      int hi = 1;
      int lo = 1;
      long value = (((long)hi) << 32) | lo;              // CS0675
      // try the following line instead
      // long value = (((long)hi) << 32) | ((uint)lo);   // correct
   }
}
```

Compiler Warning (level 3) CS1700

程序集引用 Assembly Name 无效，无法解析

此警告指示未正确指定特性（如 InternalsVisibleToAttribute）。

有关更多信息，请参见友元程序集（C# 和 Visual Basic）。

下面的示例生成 CS1700。

```
// CS1700.cs
// compile with: /target:library
using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("app2, Retargetable=f")]   // CS1700
[assembly:InternalsVisibleTo("app2")]   // OK
```

Compiler Warning (level 4) CS0429

检测到无法访问的表达式代码

每当无法访问代码中某个表达式的一部分时，便会发生此错误。在下面的示例中，条件 false && myTest() 满足此条件，因为 myTest() 方法因 && 运算符左边始终为 false 这一事实而永远不会被计算。只要 && 运算符将 **false** 语句计算为 false，它便会停止计算，并且永远不会计算右边。

下面的代码生成 CS0429。

```
// CS0429.cs
public class cs0429
{
    public static void Main()
    {
        if (false && myTest())  // CS0429
        // Try the following line instead:
        // if (true && myTest())
        {
        }
        else
        {
            int i = 0;
            i++;
        }
    }

    static bool myTest() { return true; }
}
```

Compiler Warning (level 4) CS1591

缺少对公共可见类型或成员“Type_or_Member”的 XML 注释

指定了 /doc 编译器选项，但有一个或多个构造没有注释。

下面的示例生成 CS1591：

```
// CS1591.cs
// compile with: /W:4 /doc:x.xml

/// text
public class Test
{
   // /// text
   public static void Main()   // CS1591, remove "//" from previous
   {
   }
}
```

Compiler Warning (level 4) CS1610

无法删除用于默认 Win32 资源的临时文件“file”-- resource

在使用 /win32res 编译器选项时以及当 **%TEMP%** 目录没有 DELETE 权限时，该警告指示编译器未能删除它创建的临时文件。

确保对 **%TEMP%** 目录有读/写/删除权限。

如有必要，可以手动删除这些文件，这对 C# 或您的任何程序没有损害。

C# 语言规范

C# 语言规范是 C# 语法和用法的权威来源。此规范包含有关语言所有方面的详细信息，包括 Visual C# 的文档中并未涵盖的许多要点。

可以从 Microsoft 下载中心下载此规范。如果已安装 Visual Studio 2013，则你还可以在计算机上的 Program Files (x86)/Microsoft Visual Studio 12.0/VC#/Specifications/1033 文件夹中查找此规范。但 Visual Studio Express 2013 的安装不包含此文件。

请参阅

C# 参考

C# 编程指南